Inauguraldissertation zur Erlangung der Doktorwürde der Neuphilologischen Fakultät der Ruprecht-Karls-Universität Heidelberg

Global Inference and Local Syntax Representations for Event Extraction

vorgelegt von

Viktor Alexander Judea

Referent: Prof. Dr. Michael Strube
Korreferent: Prof. Dr. Anette Frank
Einreichung: März 2019
Disputation: 20. Mai 2021

Ich widme die vorliegende Arbeit
meiner Frau Bianca und meinen Kindern Luca und Miriam.

Ich setzte den Fuß in die Luft,
und sie trug.

Abstract

Event extraction is the task of automatically finding events in texts. It is an important step towards automatic text understanding because events not only describe *what* happens, but also assign roles to participating entities. Events are complex semantic structures. Finding events in an information extraction setting consists of finding a word which indicates the event on the lexical surface, called the trigger, and a set of arguments, entity mentions which play a role in the event, along with the roles they play.

Many event extractors published to date capture only intra-sentential contexts and rely on shallow features like the neighbor words and immediate dependency relations. This thesis is concerned with expanding the information available to an event extractor. We propose a method to make the global (document-wide) context available to the decoding process of a local (intra-sentential) state-of-the-art event extractor. The resulting system shows the best evaluation results to date (summer 2018). Our system improves overall performance because it can improve the identification and classification of triggers. We could not devise successful features for a global event argument detection. This is the starting point for the second part of the thesis.

We investigate the argument prediction performance of the base system we improve with global inference and find that the performance is strongly tied to the distance of a potential argument: Arguments which are closer to the trigger can be predicted much more reliably than arguments which are far from it. We hypothesize that this effect is due to data sparseness – a system can learn to predict arguments close to the trigger better because it involves less divergence in the words and the relevant syntactic relations. A system which has the ability to represent syntax structures of arbitrary length and independently of their prominence during training has an advantage. We show that such a system has indeed a considerably better argument classification performance compared to the baseline. However, this system operates under laboratory conditions: Because we want to evaluate the performance of our system on argument

predictions in isolation, without interference of noisy triggers, we assume that triggers are already given.

Finally, we extend this system to a full event extractor which also predicts triggers. This final system also depends on syntax structures – where we used shortest dependency paths for laboratory conditions, we now operate on entire dependency graphs instead and perform trigger and argument extraction based on these structures. Our final event extractor also provides a common event extraction framework (same preprocessing, same infrastructure) to directly compare two graph encoding methods, namely Graph Convolutional Networks and tree-shaped Long Short-Term Memory Networks, in terms of their ability to provide useful information for event extraction. We again find that syntax representations do help event extraction, even with predicted and noisy triggers. Additionally, we again show that improving trigger classification recall has a great influence on argument classification performance – a method can improve argument classification performance solely by improving trigger recall.

We also propose various methods to combat the small amount of training data we have. We make the training process of neural networks more stable by averaging parameters across training epochs. Additionally, we train our final system with bagging – a method which uses multiple versions of the training data to produce an ensemble of predictors. Finally, we propose a new undersampling method to directly address the high class imbalance during trigger prediction training.

In the last years, neural networks had a renaissance in the form of Deep Learning. Two factors which led to this development are new random initialization methods which considerably increase learning ability, and faster training on Graphics Processing Units. Both factors introduce randomness into the training process, which has a profound impact on the reliability of scientific evaluations – the same network with the same hyperparameters can produce different, statistically significant evaluation results when training it multiple times. Whenever we use deep learning methods, we train five models, evaluate five times and report average results and sample standard deviations in order to report more reliable results.

Zusammenfassung

Eventextraktion bezeichnet die Aufgabe, automatisch wichtige Ereignisse (Events) in texten zu finden. Als solche ist die Aufgabe ein unverzichtbarer Schritt hin zum automatischen Textverstehen. Events sind komplexe semantische Strukturen: Sie bestehen aus einem Wort, welches das Event an der Textoberfläche anzeigt, genannt der Trigger und einer Anzahl von Argumenten, Erwähnungen von Entitäten welche eine Rolle in den Events spielen, zusammen mit den Rollen, die sie einnehmen.

Viele bislang publizierte Event Extraction-Systeme operieren nur satzintern und nutzen flache Features wie Nachbarwörter und eigene Dependenzrelationen. Diese Arbeit befasst sich mit der Erweiterung der Information, die einem Event Extraction-System zur Verfügung steht. Wir präsentieren eine Methode, die den dokumentweiten Kontext einem ansonsten nur lokal (satzintern) arbeitenden Event Extraction-Programm verfügbar macht. Das daraus resultierende neue System zeigt die besten bisher gezeigten Evaluationsergebnisse (Stand: Sommer 2018). Unser System verbessert die Gesamtperformanz weil es die Identifizierung und Klassifizierung von Triggern verbessern kann. Wir konnten jedoch keine erfolgreichen Features konstruieren, die eine globale (dokumentweite) Verbesserung der Detektion von Argumenten ermöglichen. Dies ist der Startpunkt für den zweiten Teil der vorliegenden Arbeit.

Wir untersuchen die Argumentextraktions-Performanz desjenigen Basissystems, welches wir mit globaler Inferenz verbessern und finden, dass sie stark zusammenhängt mit der Distanz eines potentiellen Arguments: Argumente, welche näher am Trigger stehen, können viel besser erkannt werden als solche, die weit weg stehen. Wir stellen die Vermutung auf, dass dies spärlichen Trainingsdaten geschuldet ist – ein System kann besser lernen, dem Trigger nahe Argumente zu erkennen, weil weniger Variabilität hinsichtlich relevanter Wörter und Sytaxrelationen besteht. Ein System, welches Syntax auf eine von der Länge der Struktur und ihrer Trainingsverfügbarkeit unabhängigen Weise repräsentieren kann, hat hier einen Vorteil. Wir zeigen, dass ein solches System in der Tat zu einer besseren Performanz der Argumentklassifikation führt.

Jedoch operiert dieses System unter künstlichen Bedingungen. Es nimmt an, dass Trigger schon gegeben sind, weil wir die Systemperformanz nur hinsichtlich der Argumentfindung untersuchen wollen, ohne Interferenzen von verrauschten Triggervorhersagen. Wir erweitern dieses System zu einem vollständigen Event Extraction-System. Dieses finale System operiert direkt auf Syntaxstrukturen – wo wir vorher kürzeste Dependenzpfade nutzten, repräsentieren wir nun auf ganzen Dependenzgraphen und führen Trigger- und Argumentvorhersagen basierend auf diesen Repräsentationen aus. Wir stellen ein gemeinsames Framework auf (mit einheitlicher Vorverarbeitung und Infrastruktur), um zwei Methoden zur Graphrepräsentation, nämlich Graph Convolutional Networks und baumförmige Long Short-Term Memory Networks hinsichtlich ihrer Nützlichkeit für Event Extraction zu vergleichen. Wir zeigen, dass solche Repräsentationen Event Extraction helfen, dieses Mal auch mit verrauschten Triggervorhersagen. Außerdem zeigen wir, dass ein System, welches nur den Recall von Triggervorhersagen erhöht, auch eine wesentlich bessere Argumenrvorhersage erreichen kann.

Wir stellen auch mehrere Methoden vor, um mit der geringen Trainingsdatenmenge besser umzugehen. Wir machen den Trainingsprozess Neuronaler Netzwerke stabiler, indem wir ihre Parameter über verschiedene Trainingsepochen hinweg mitteln. Zusätzlich trainieren wir unser finales System mit Bagging, einer Methode, welche unterschiedliche Versionen der Trainingsdaten verwendet, um ein Prädiktorenensemble zu trainieren. Schließlich schlagen wir noch eine neue Methode zur Unterabtastung von Trainingsdaten vor um direkt das Ungleichgewichtsproblem während des Triggervorhersage-Trainings anzugehen.

In den letzten Jahren haben Neuronale Netzwerke eine Renaissance in der Form von Deep Learning erlebt. Zwei Faktoren, die zu dieser Entwicklung beigetragen haben sind neue Methoden zur zufälligen Initialisierung, welche die Lernfähigkeit stark erhöhen und schnelleres Training auf Grafikprozessoren (GPUs). Beide führen Zufallsprozesse ins Training ein, was wiederum profunde Auswirkungen auf die Verlässlichkeit von Evaluierungen hat – das gleiche Netzwerk kann statistisch signifikant unterschiedliche Ergebnisse liefern, wenn es mehrfach trainiert wird. Wann immer wir Deep Learning-Methoden benutzen, trainieren wir fünf Modelle, evaluieren fünf Mal und berichten durchschnittliche Zahlen zusammen mit Standardabweichungen um verlässlichere Evaluationszahlen zu erhalten.

Danksagungen

Zuerst möchte ich meinem Doktorvater Michael Strube danken. Ich hatte mich im Verlauf der Doktorarbeit zwei Mal verrannt – einmal bei der Entwicklung der globalen Inferenz und einmal beim Nachbau eines schon publizierten Systems. Das erste Mal hat er mich ermuntert, meine immer komplizierter werdende Methode auf eine möglichst einfache Version zu reduzieren. Es hat sich herausgestellt, dass diese weniger komplexe Version robuster und besser arbeitete als alle komplexeren Varianten vor ihr. Das zweite Mal hat er mir die Zeit gegeben, mich vollständig zu verlieren in eine Arbeit, die am Ende doch fruchtlos blieb. Auf meinem Irrweg habe ich Lektionen gelernt, die meine persönliche und akademische Entwicklung stark beeinflusst und positiv geprägt haben. Hätte er dem Irrweg frühzeitig ein Ende gesetzt, wären mir wichtige Erfahrungen verwehrt geblieben.

Michael hat mir und meinen Kollegen gezeigt, wie wichtig es ist, die eigene Arbeit zu präsentieren und sie der Kritik anderer auszusetzen. Er hat uns auch gezeigt, wie wichtig kritisches Denken und das Interesse an anderen Forschungsgebieten ist. Ich bin dankbar, Michael als Doktorvater zu haben. Ich habe durch ihn viel mehr als nur Fachliches dazugelernt.

Der größte Teil der vorliegenden Arbeit ist am Heidelberger Institut für Theoretische Studien (HITS) entstanden. Ich wurde durch ein Promotionsstipendium und später durch eine Anstellung in meinem Promotionsvorhaben unterstützt. Ich möchte dem HITS und der Klaus Tschira Stiftung dafür danken.

Weiterhin möchte ich meinen Kollegen danken, vor allem Sebastian Martschat, Yufang Hou und Nafise Sadat Moosavi, die Teile der vorliegenden Arbeit gelesen und mit hilfreichen Kommentaren verbessert haben. Mit Sebastian habe ich viele Stunden am Kaffeeautomaten verbracht und über alles geredet, von Filmen bis hin zu aktueller Forschung. Mohsen Mesgar und ich haben unsere neuronalen Modelle in langen Diskursen am Whiteboard verbessert. Nafise, Yufang, Benjamin Heinzerling, Angela Fahrni, Daraksha Parveen und kurzzeitig auch Feifei Peng haben meinen Aufenthalts am HITS unvergesslich gemacht.

Einen ungewöhnlichen Dank möchte ich an Frau Angelika Taudte, Herrn Peter Lenser und Herrn Bruno Weber richten. Frau Taudte hat mich als erste Lehrkraft überhaupt auf mein Potential aufmerksam gemacht. Herr Lenser wiederum hat mir in der Oberstufe die Informatik und Herr Weber die Mathematik näher gebracht. Mein Gefühl sagt mir, dass ich ohne diese drei einen anderen, viel weniger interessanten Lebensweg eingeschlagen hätte.

Zuguterletzt möchte ich meinen Freunden und meiner Familie für die Hilfe, Geduld, die Gespräche und das viele Mutmachen danken. Vor allem meiner Mutter gilt ein besonderer Dank für die Unterstützung im Studium und auch später in der Promotion.

Contents

1 Introduction

A large amount of our knowledge is encoded as unstructured information, mainly in the form of texts. Natural Language Processing (NLP) research creates computational methods and systems which can access this knowledge, from the meaning of a word to the meaning of a discourse. The ultimate goal of NLP is text understanding: to automatically and reliably infer structure (and thus meaning) in unstructured texts. An important and popular task in this endeavor is event extraction, the task of automatically predicting 'what happened, to whom, when, and where'.

The exact definition of an event is part of rich philosophical debates.[1] In this thesis, we adopt a task-centric and pragmatic view of events: An event is identical to a special kind of text annotation consisting of a word which indicates the event, called the trigger, and zero or more mentions of entities which play a role in the event, called arguments. Event extraction is the task of automatically producing these annotations in texts.

Another view on events, or event annotations, is that of a template. In event extraction, we are given a finite set of templates we want to automatically fill, one for each event type. An event template consists of a *trigger* and a set of *roles* an entity (a person, organization, etc.) can play in the event. Triggers and arguments have a label; the trigger label is the kind of event which is being triggered; the argument label is the role the respective entity mention plays in the event.

In the following example, the word "returned" is a trigger and indicates a transportation event. "Bush" and "Ireland" are two entity mentions, namely the mention of a person and of a location, which also play a role in the event: They are the entity being moved and the origin of the transport.

(1) Bush returned from a summit in Ireland.

[1] See, e.g., Davidson (1969); Mourelatos (1978); Moens and Steedman (1988); Pustejovsky (1991) and a very brief discussion in Section 2.1.

Event extraction seeks to automatically fill the abstract 'transport template', which consists of a trigger placeholder and all the roles the respective event has (agent, artifact, origin, destination, time, and vehicle), with concrete occurrences in the text (transport trigger: "returned", artifact: "Bush", origin: "Ireland", other arguments unfilled).

Event extraction as template assignment corresponds to the setting used in the 2005 Automatic Content Extraction conference (henceforth, ACE 2005). ACE defines event extraction in terms of three sub-tasks: *Entity mention detection*, finding mentions of predefined entity types like persons and organizations; *event trigger identification and classification*, finding words which indicate an event and predicting which event they trigger; and *event argument identification and classification*, finding the entities playing a role in an event together with their roles. Entity mention detection is commonly omitted, which enables research to better focus on the core problems in event extraction.

In this thesis, we treat event extraction as a prediction task: Given a text, find event triggers along with the event types they evoke, and find all their arguments along with the roles they play. In order to carry out such a task, we make extensive use of *machine learning methods* which we describe in more detail below.

Section 1.1 states our motivation for this thesis and the research questions we address. Section 1.2 discusses the contributions of our work and Section 1.3 states our publications which underlie this thesis. Finally, Section 1.4 mentions all machine learning methods we use. However, they are formally introduced and defined in the chapters where they occur in first.

1.1 Motivation and Research Questions

ACE events never cross sentence boundaries. If an event argument is not mentioned in the same sentence as the trigger, it is not part of the respective event mention. This characteristic of ACE enables a more concise representation of event mentions and it simplifies the automatic extraction, but it also leads most research to predict events based on single sentences, ignoring the document-wide context. Research which incorporates entire documents relies either on rigid, hand-crafted rules or multiple machine learning models which cooperate in complex ways. Furthermore, a great part of publi-

cations focus on new prediction methods and use similar information to carry out the actual predictions, ignoring important intra-sentential information like syntax graphs.[2]

The motivation for this thesis is based on two observations: (1) Often, there is not enough information to successfully predict events based on single sentences. (2) Most event extractors are not able to fully grasp the available intra-sentential information, especially on the syntactical level. The two points correspond to the two main research questions in this thesis.

1. Can we enable a state-of-the-art intra-sentential event extractor to easily access information from the entire document during prediction?

2. Can we learn a useful and flexible syntax representation which is able to cope with syntactical structures never seen during training?

We answer Question 1 in Chapter 3 where we introduce a global inference method which enables a local system to draw information from similar event assignments throughout a document. We show that this considerably enhances performance.

However, we also find that improving argument classification performance with global information is difficult once the base system reaches a certain reliability in its argument predictions. Chapters 4 and 5 are dedicated to answer Question 2. To the best of our knowledge, we are the first to analyze argument identification and classification performance per se.

Chapter 4 starts with an analysis of our base system's argument prediction performance. We show that syntactical distance is a crucial factor in predicting arguments. The farther away a potential argument is from the trigger, the more difficult it becomes to predict it correctly. This is not only true for individual argument assignments, but also for entire argument types. `Victim` arguments for example tend to be expressed nearer to their trigger compared to `Place` arguments. Consequently, our base system predicts them considerably better (+18 F_1 points). We hypothesize that a system can better predict arguments close to their trigger because it has seen most syntax structures which connect the two during training. Most systems either use local syntax information (like the subject of a word) or they decompose a syntax graph (the

[2] Syntactical information is of course an important part of event extractors, but they use it either in the form of local word-to-word dependencies, or in the form of categorical syntax paths. Categorical features are the information used in feature-based machine learning algorithms. They have the drawback that new features, e.g., syntax paths never seen during training, do not have a meaning for prediction.

syntactical analysis of a sentence) in more simple categorical features. This has the disadvantage that syntactical constructions never seen during system training cannot be used for prediction. Question 2 directly addresses this issue: How can we devise better representations of syntax structures which can assign meaning to *any* syntactical structure, even those never seen in training? We show that, when inspected in isolation without noise from wrong trigger prediction, such a representation improves event argument classification performance considerably. In Chapter 5, we extend the representation from linear syntactical paths to general syntax graphs and show that such complex structures can improve the performance of event extractors in general.

1.2 Contributions

Chapter 3 proposes a global decoding method for event extraction (Section 3.2) which can enable an intra-sentential event extractor to access information from the entire document during prediction. We use it with two versions of an intra-sentential event extractor in two different settings (Sections 3.3.1 and 3.3.2). Furthermore, we introduce new feature types to event extraction (Section 3.2.2).

In Chapter 4, we present an analysis of event argument classification performance. To the best of our knowledge, we are the first to investigate this aspect in isolation (without interference from trigger predictions). One main finding is that event argument performance is strongly influenced by trigger predictions – we show that improving trigger prediction recall usually leads to improved argument prediction performance (Sections 3.3.1 and 5.7.2). This is true even if the argument prediction *per se* is not better. Increasing trigger recall enables a system to find more arguments, which, given a good argument prediction mechanism, usually leads to better argument performance.

In Chapter 4, we find that argument prediction performance is strongly connected to syntactical distance – arguments which stand far from their triggers are less likely to be predicted correctly. This observation leads us to the second subject in this thesis: a representation which can encode arbitrary syntax structures, even those which were never encountered during training. In Chapter 4 we show that such a representation improves argument predictions considerably when investigated in isolation.

In Chapter 5, we investigate two more complex syntax encoders (Graph Convolutional Networks and tree-shaped Long Short-Term Memory Networks) with respect to

their usefulness for event extraction. In this chapter, we also address the problem that ACE 2005 provides only little training data and use Bootstrap Aggregating (bagging) to train our models.

The final contribution we want to mention here is that Chapters 4 and 5 address reliability and comparability of scientific evaluations. Section 2.5 discusses two of the issues involved in this complex area: Sometimes, a better preprocessing can considerably improve performance even if the involved system does not change. This leads us to the practice that we compare systems and settings with identical preprocessing in our work, e.g., when we compare our local and our global event extractors, or when we compare the different syntax encoders. We cannot however test other published systems in this way because we have, with the exception of our base system in Chapter 3, no other runnable event extractor. The second point we discuss is related to indeterministic training. Training deep learning methods involves randomness. This has a profound impact on determinism: The same method produces different models given the exact same input, and these models in turn may perform very differently in the same test set. A way to increase reliability of evaluations is to report average evaluation metrics across multiple training and testing rounds. However, it is common practice in the ACE event extraction literature (and in other computer science publications) to report only one training and testing round. This raises questions about the reliability of evaluations. Whenever we use indeterministic training, we report the average of 5 training and testing round, as well as sample standard deviations where appropriate.

1.3 Publications

This thesis is in large parts based on three of our publications. In Section 2.3.3 we mention Judea and Strube (2015) when we talk about tasks which are structurally identical to event extraction, most notably frame-semantic parsing.

Judea and Strube (2016) is the foundation of Chapter 3, especially for the setting with predicted entity mentions. However, we extend this study by also exploring the gold entity mention setting.

Judea and Strube (2017) corresponds to the argument performance analysis and the dependency-path encoder we present in Chapter 4. Therefore, this publication

also constitutes the foundation of Chapter 5 where we explore syntax representations of broader syntactical structures.

We publish accompanying code for Chapter 4 on the heiDATA servers for a more persistent, long-time archiving with the DOI 10.11588/data/CZZEKX (Judea, 2021a).

Accompanying code *and* models for Chapter 5 are also published on heiDATA with the DOI 10.11588/data/Z1RKOI (Judea, 2021b).

Furthermore, we publish both *code repositories* (without models) on GitHub (`https://github.com/m-alexj/argumentor.git` and `https://github.com/m-alexj/eventor.git` respectively).

1.4 Methodological Overview

This section briefly describes the methods we use and references them to the chapters and sections where they are introduced and defined.

We heavily rely on supervised machine learning to carry out event extraction. We have a set of documents with manually produced event annotations. Machine learning algorithms analyze the annotations and produce models which abstract from the information they were produced on, making them able to annotate events in new sentences. The machine learning methods we use can be divided into two broad groups: *feature-based* and *Deep Learning*. The former relies on hand-crafted features which decompose the problem into single characteristics, e.g., the left and right lexical context of a word (when computing event triggers) or the syntactical relations of a head noun (when computing event arguments). Deep Learning methods on the other hand learn latent representations of the problem, without the need for manual feature engineering. They only rely on an input, an architecture of different processing layers, and an output. During learning, the weights in the architecture are formed in a way that the input likely results in the desired output.

Chapter 3 uses a feature-based machine learning method, namely the **structured perceptron** (Collins, 2002; Huang et al., 2012). Section 3.1 introduces and formalizes the structured perceptron in the context of event extraction. In Section 3.2, we propose a multi-pass inference method to incorporate global (document-wide) decisions into the local (intra-sentential) structured perceptron we use.

In terms of machine learning methods we have a turning point from Chapter 3 to Chapters 4 and 5. This turning point coincides with the recent popularity of **Deep**

Learning (DL) in computer science. DL methods do not rely on manually engineered features; they only require an input and a known output – training procedures ensure that the system automatically learns to produce the right output, without the necessity of telling it explicitly what to pay attention to.

DL offers an important advantage for our work: it can produce representations of arbitrary syntax structures, from linear dependency paths to general dependency graphs. Instead of decomposing such structures into features and learning weights for them, DL methods embed them more directly into a high-dimensional, continuous space such that similar structures (with respect to event extraction) stand close together. We investigate the use of such syntax representations for event argument classification in particular (Chapter 4) and for event extraction in general (Chapter 5).

In Chapter 4, we introduce **Long Short-Term Memory** networks (LSTMs) (Section 4.3.3). LSTMs have the ability to encode an arbitrarily long sequence into one fixed-size vector. In Chapter 5, we extend simple LSTMs to general graphs. We test the use of syntax graph encoding methods for event extraction, namely the use of **Graph Convolutional Networks** (Section 5.5.3) and **tree-shaped LSTMs** (Section 5.5.4). Section 4.3.4 introduces and formalizes **Convolutional Neural Networks** (CNNs). CNNs efficiently learn patterns from their inputs, e.g. to recognize specific objects in images – we use them to learn patterns in the lexical context of potential event arguments.

We also use auxiliary methods. Most notably, we use **parameter averaging** in each chapter. Instead of predicting event triggers and arguments using the latest weights, we average them with previously obtained weights. This has the advantage that weights which oscillate heavily during training are smoothed out, while weights which are nearly constant remain unaltered. This method was introduced in Collins (2002) and further formalized in Huang et al. (2012) to considerably improve the performance of perceptrons, and to bring them in a par with more sophisticated feature-based learning algorithms like Support Vector Machines (Hearst et al., 1998). We also use parameter averaging for our DL systems (Section 4.3.8).

2 ACE Event Extraction: Task and Data

In this thesis, we work with the popular event schemes used by the Automatic Content Extraction (ACE) program and its subsequent version, the Text Analysis Conference (TAC)[1]. The ACE event scheme was developed by the US-American National Institute of Standards and Technology (NIST). TAC annotations are based on a lighter scheme, namely on 'Entities, Relations, and Events' (ERE, Song et al., 2015) developed in the Deep Exploration and Filtering of Text (DEFT) program, financed by the Defense Advanced Research Projects Agency (DARPA), a US agency. Both schemes have in common that they view event extraction as an information extraction task. In this chapter, we present the annotation schemes and put them in a broader context, including a short discussion of viewing the task in computational and scientific terms.

We briefly establish events as philosophical entities (Section 2.1) before we adopt a pragmatic view and define an event to be identical to the ACE annotation of an event. Section 2.2 describes the ACE annotation scheme in detail. Section 2.3 locates event extraction in the space of Natural Language Processing tasks and mentions other event annotation schemes. Finally, Sections 2.4 and 2.5 describe event extraction as a computational task and as a scientific pursuit, respectively.

2.1 Establishing Events

The exact definition of an event is subject to ongoing philosophical debates (Davidson, 1969; Mourelatos, 1978; Moens and Steedman, 1988; Pustejovsky, 1991, i.a.). Davidson (1969) for example replaces the question 'what are events' with the question 'when are two events identical'. He proposes to treat events the same as entities – they are

[1] We discuss and use the event trigger annotations in the TAC 2015 data. The TAC argument annotations differ from their ACE counterparts in a fundamental (and for us unusable because out of scope) way: They are based on coreference chains and can occur anywhere in a document.

located in space and time and have attributes which specify them further; when they are used in sentences, one can define that two event mentions are coreferent (refer to the same event) when they refer to the same 'event-entity'. This view makes events easier to understand because they are treated similar to entities like cars and birds. However, it also leads to problems on a conceptual level because some events do not fit well in this entity-centric view. Two events can share the same point in spacetime for example, or continuously blend into each other.

The formal notion of events is deeply rooted in language philosophy, and beyond the scope of this thesis. We will not define events formally, but we will report and adopt the notion of events used in creating the datasets we operate on. These notions are pragmatic in nature and thus leave out many important philosophical and linguistic aspects, but they help to define events in such a way that they become intuitively understandable by humans (which is important for manual annotation) and processable by software (which is the ultimate goal of the effort). In the following, all quotes are from the ACE 2005 event annotation guidelines (Linguistic Data Consortium: Events, 2005).

> An Event is a specific occurrence involving participants. An Event is something that happens. An Event can frequently be described as a change of state.

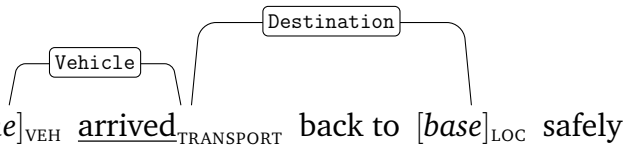
Events connect entities, times, and places and therefore constitute a higher semantic level than these categories. There is a basic distinction between events and event *mentions*. The latter are concrete mentions of an event in texts. Two event mentions are connected by a coreference link if they refer to the same event, potentially revealing or highlighting different aspects of the event. It is possible for example that one mention talks about the attacker of an event, while another mention talks about the victims of the same event without mentioning the attacker. Knowing which mentions refer to the same event allows to properly aggregate all the information reported in a document, or even a document collection. In this thesis, we are exclusively concerned with the extraction of event mentions and leave the interesting and fundamental problem of event coreference aside.

2.2 Describing ACE Events

Above, we 'defined' events following the ACE annotation guidelines. In this section, we want to describe the underlying annotation scheme and the ACE data. We also describe the annotation of the Text Analysis Conference (TAC) 2015 because they are strongly related to ACE and we use them for evaluation in Chapter 3.

An *event template* consists of a type, a set of specific roles, and placeholders for a trigger and arguments. For example, the movement template has the type MOVEMENT and the roles `Agent`, `Artifact`, `Origin`, `Destination`, `Time`, and `Vehicle`, as well as the mentioned placeholders.[2] Event templates are organized in eight broad categories, namely LIFE, MOVEMENT, TRANSACTION, BUSINESS, CONFLICT, CONTACT, PERSONNEL and JUSTICE. The word which "most clearly" (Linguistic Data Consortium: Events, 2005) indicates an event is the trigger. An argument is a role filler. More precisely, an argument of an event is an entity mention which plays one of the roles predefined by the respective event template. This means that ACE ignores all event types which are not of interest, and it ignores all entities which do not play one of the predefined roles in an event.

An *event* is a specific instance of a template. It consists of a trigger and arguments, each associated with a type. The trigger type is the event type. An argument type is the role the respective entity mention plays in the event. An *event mention* is the occurrence of an event in the text. To be more concise however, we will mostly talk about 'events' and only write 'event mentions' where the distinction is necessary. Consider the following example of an event mention.

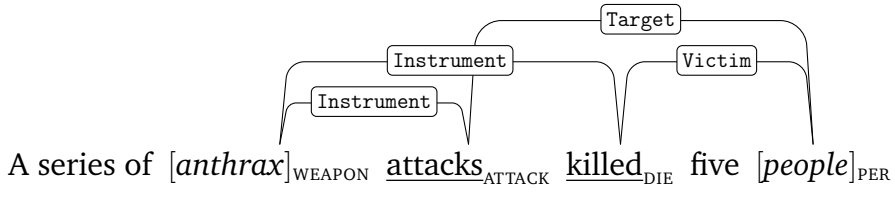
(2) 

In Example (2) we can find one TRANSPORT event triggered by "arrived" with the two arguments "plane" and "base". "Plane", the mention of a vehicle entity, fills the `Vehicle` role. "Base", the mention of a location entity, fills the `Destination` role. In order to be more concise, we will say that "arrived" is a TRANSPORT trigger (we omit

[2] In this thesis, event types are always in SMALL CAPS and roles in `Typewriter` font. Entity mentions stand in square brackets, the entity type comes afterwards as a subscript: $[he]_{\text{PER}}$ for example is the mention of "he", a PER (person) entity.

the event supertype), and that "plane" and "base" are `Vehicle` and `Destination` arguments, respectively. Other mentions of this event might specify additional arguments in other places of the document. The totality of such arguments constitutes the abstract event, which in turn is an instance of the transport event template. We will discuss all categories introduced so far in more detail in the following subsections.

In (2) we have only one event. The next example contains multiple events and shared arguments.

(3)


In (3), "attacks" triggers an ATTACK event and "killed" triggers a DIE event. All of the arguments in (3) are shared by the two events. "Anthrax" is an `Instrument` to both. "People" fills different roles in the two events: It is the `Target` of the ATTACK, but the `Victim` of the DIE event.

We will now give more detailed explanations of the three components of ACE event extraction: event mentions, triggers, and arguments. Afterwards, we discuss entity mentions and some of the difficulties and errors with ACE annotations we observed during development.

2.2.1 Event Mentions

We introduced event mentions above and demarcated them from events. Here, we describe additional information associated with event mentions in the ACE data.

ACE and subsequent annotation schemes define an *extent* of the event, which is equal to all tokens which include the trigger and all arguments. The *scope* of an event is a sentence, meaning that ACE events do not cross sentence boundaries. For 'Entities, Relations, Events' (ERE) in contrast, the scope is the entire document, meaning that events *do* cross sentence boundaries. An argument in TAC for example can occur anywhere in the document. Finding TAC arguments makes it necessary to compute entity mentions coreference, a hard NLP problem in itself.

There is more information associated with an event: *Polarity* refers to the actuality of the event. Positive polarity means the event actually happened, negative polarity refers

to the opposite case. This is important because ACE allows the annotation of 'non-real events', i.e. hypothetical, commanded/requested, threatened/proposed/discussed events (encoded as the event's *modality*). *Tense* indicates if the event is a past or future event with respect to the document's publication time. Finally, *Genericity* indicates if an event is generic or specific. Specific events are those happening at a specific point in spacetime; all other events (e.g., repetitive) are generic.

2.2.2 Event Triggers

Event triggers are words or phrases that express an event occurrence within a sentence. Identifying triggers is an essential part of both ACE and TAC. In ACE, triggers always have a head consisting of only one word. In TAC, the trigger may consist of multiple (contiguous) words.

For a better understanding of triggers, we analyze part-of-speech (POS) distributions in ACE and TAC, more specifically in the training set we use throughout this thesis. We apply a POS tagger to both sets and gather statistics about the trigger head words. Table 2.1 reports the results. We discuss triggers from each part-of-speech and give examples afterwards.

	ACE05		TAC15	
part-of-speech	%	#	%	#
verb	49.1	2169	51.4	3027
noun	45.6	2014	43.2	2544
adjective	3.1	138	3.7	216
pronoun	0.9	42	0.6	30
other	1.3	56	1.1	70

Table 2.1: The distribution of the four most frequent part-of-speech tags for event triggers in the ACE 2005 and the TAC 2015 training set. '%' refers to fractions, '#' to frequencies.

As we can see in Table 2.1, the part-of-speech tag distribution is similar in both datasets. Around 50% of all triggers are verbs. Consider the following examples.

(4) Orders went out today to deploy 17,000 U.S. Army soldiers ...

(5) At least 19 people were killed and 114 people were wounded ...

With around 45%, nouns are the second most frequent POS category for event triggers. Most of them are verb nominalizations, many of which also occur as verbs in the training set.

(6) After Christmas, I got a call from the wife of one of my former bosses ...

(7) The toughest fight, though, may lie ahead in the heart of the Iraqi capital.

Besides common nouns, the noun category also includes proper nouns. Most are tagging errors, e.g., capitalized triggers like 'Murdered' which were confused with proper nouns by the automatic POS tagger. Correct proper noun triggers include salient military events ('Operation Iraqi Freedom', 'World War II') and dates which are used metonymically ('September 11'). One group which does not occur in ACE or TAC are places where a salient event occurred and which are used metonymically, e.g., "Fukushima". It is an interesting problem to analyze metonymically used triggers in particular. Unfortunately, they are very rare in ACE and TAC.

(8) Famed World War II reporter Ernie Pyle

Pronouns are responsible for less than 1% of event triggers. This part-of-speech includes only 'it' and 'them' if they refer to another event mention in the text. Consider the following example.

(9) A student fatally shot a principal before killing himself this morning. It happened in the cafeteria of red lion area junior high school about 30 miles southeast of Harrisburg.

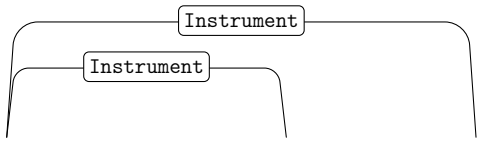
In (9), "it" is a DIE trigger and refers back to the event expressed by "killing".

2.2.3 Event Arguments

As we outlined above, event templates specify a set of roles. By extension, each event has the same set of roles to fill as the respective template. Most of the roles are template-specific. For example, ATTACK events have the roles `Attacker`, `Target` and

Instrument to fill, whereas DIE events have the roles Agent, Victim and Instrument to fill. Common to all are the roles Time and Place.[3]

In ACE, roles can only be filled by entity mentions.[4] If a role filler exists, it is called an argument. Therefore, we will use the term 'argument type' synonymously to 'role'. Please note that one role can be filled by zero or more fillers; more than four fillers however are very rare. In Example (10), both arguments ("bombs" and "missiles") fill the same role (Instrument) of the ATTACK event.

(10) 

... those sorties will be strike sorties using Bombs and Missiles ...

Arguments can be further subdivided into two classes: Participants and attributes. Participants are persons (*per*), organizations (*org*), geo-political entities (*GPE*), facilities (*fac*), locations (*loc*), vehicles (*veh*) or weapons (*wea*). Attributes are subdivided into event-specific and general. Event-specific are the attributes Crime and Sentence (JUSTICE events), and Position (PERSONNEL events). General atrributes are Place and Time, which apply to every event type.

The event argument annotation for TAC 2015 follows a different intention: Arguments are annotated per event (as opposed to per event mention), meaning that the arguments of an event are the most specific role fillers which occur *anywhere in the document*. To illustrate this, consider the following text from the TAC argument linking task description draft (from July 14, 2015):

(11) A separatist group called the *Kurdistan Freedom Falcons (TAK)* claimed responsibility for an explosion *late on Monday* which wounded *six people*, one of them seriously, in an Istanbul supermarket. Istanbul governor Muammer Guler told Anatolia news agency the explosion in *the Bahcelievler district* of Turkey's largest city injured six people. The agency said *15 other people* had been hurt. "We consider the explosion that took place tonight in an Istanbul supermarket to be a response to the barbaric policies against the Kurdish people

[3] For MOVEMENT events, Origin and Destination very often substitute Place.
[4] In this thesis, we expand the notion of an entity, see Section 2.2.4.

Two events are mentioned multiple times in (11), an ATTACK and an INJURE event. We focus on the former. The event is indicated by multiple trigger words ("explosion"). Its arguments are spread across multiple sentences. For example, the `Attacker` ("Kurdistan Freedom Falcons (TAK)") is in another sentence than the two `Targets` "six people" and "15 other people". In the case of competing fillers (e.g., "late on Monday" vs. "tonight") the most specific one is selected.

2.2.4 Entity Mentions

As mentioned above, roles can only be filled by entity mentions (under a broader notion of 'entity', see below). When we write 'entity mention', we mean the seven ACE entity types as well as times, numbers, and event-specific attributes (crimes, legal sentences, and employee positions) for the sake of simplicity.

One would assume that entity mention prediction is a fundamental task in event extraction. It is, however, mostly ignored in the literature. The standard setting in ACE event extraction is that entity mentions are given. Therefore, most event extractors operate in a somewhat artificial setting and cannot be used 'in the wild' to extract events.[5] In the following, we want to briefly discuss entity mentions. We start with an example.

(12) Orders went out $[today]_{\text{TIME}}$ to deploy 17,000 $[U.S.]_{\text{GPE}}$ $[army]_{\text{ORG}}$ $[soldiers]_{\text{PER}}$

We have four entity mentions in (12). "Today" is a point in time, "U.S." a geopolitical entity, "army" an organization, and "soldiers" are persons.

ACE defines two annotations for entity mentions: the extent and the head. The extent is a nominal phrase (Linguistic Data Consortium: Entities, 2005). In the example above, the extent of "soldiers" is "17,000 U.S. Army soldiers" and the head just "soldiers". The head is either the syntactic head of a nominal phrase or the full extent of a proper noun. Heads seldomly overlap. In (13) we report the extents of all entity mentions in (12).

(13) Orders went out [today] to deploy [17,000 [U.S. Army] soldiers]

The three event extractors which work with predicted entity mentions (Li et al., 2014; Yang and Mitchell, 2016, our work in Chapter 3) predict heads, not extents.

[5] Yang and Mitchell (2016) and our global event extractor in the 'predicted entity mention' setting (Section 3.3.2) are exceptions.

As mentioned above, we include the fillers of event-specific attributes like crimes in our notion of entities to be more concise. In terms of annotation however, there is an important difference between entities and these fillers: Fillers of event-specific attributes (called *values* in ACE) only have extents. This poses a serious problem for prediction. Entity heads are usually short, but value extents tend to be very long, e.g., "those attacks that killed five people and sickened 13 others". Li et al. (2014) circumvent the problem by ignoring all values. Consequently, our global event extractor in the 'predicted entity mention setting', which uses this system as its local predictor, also ignores values. Yang and Mitchell (2016) do not mention how they treat values.

2.2.5 Annotation Difficulties and Errors

Reliably annotating rich structures like event mentions in texts is demanding. In this section, we analyze some salient annotation difficulties and errors in the ACE 2005 training set, as well as principal difficulties with the annotation scheme itself.

The first difficulty we address is multiple potential triggers for one event. Consider the following example.

(14) The company was ordered to pay a fine

Here, "pay" and "fine" refer to the same event, but the ACE annotation guidelines forbid to annotate multiple triggers for one event. Instead, they specify rules which of the possible triggers to select. The annotation guidelines list a few examples of concurring triggers, but we speculate that these examples are not enough to produce annotations with high inter-annotator agreement whenever the annotators decide which of two potential triggers to keep. Often, the rules state to select some trigger over another based on its part-of-speech and if it can refer to the event by itself. In our opinion, the latter criterion is rather subjective. Whenever a noun and a verb are possible triggers, the noun is preferred if it can refer to the event by itself. In our example above, only "fine" would be annotated as a trigger.

(15) The explosion left at least 30 dead

In Example (15), a verb ("left") concurs with an adjective ("dead") for the trigger position. According to the guidelines, the adjective is preferred over the verb, but again only if it can refer to the event by itself.

Another problem arises when one word could potentially trigger multiple events. Consider the following example.

(16) The gunmen shot Smith and his son

Here, "shot" triggers an ATTACK *and* a DIE event. The ACE guidelines do not mention such cases which might introduce a source of annotation inconsistencies because annotators are free to choose one over the other. In contrast, ERE guidelines, especially the TAC guidelines, explicitly allow and encourage one word to trigger multiple events if appropriate.

We will now discuss annotation errors. Consider the following example.

(17) Police are now considering the possibility that the remains are those of Laci Peterson and her unborn child.

"Unborn" in (17) is a BE-BORN trigger. The annotation guidelines state that a BE-BORN event only occurs if an entity is born (Linguistic Data Consortium: Events, 2005). They fail to mention however that planned, commanded, or negated events are explicitly allowed, meaning that "unborn" can indeed be the trigger of a BE-BORN event. This contradiction in the guidelines causes annotator disagreement: 'Unborn' occurs 13 times in the ACE training set, 10 times mentioning the same event as in (17). However, it was annotated only twice as a trigger.

(18) ... calling on Muslims to wage jihad against the United States and its allies.

Here, "jihad" is an ATTACK trigger with "Muslims" as the `Attacker`. The phrase "wage jihad against the United States and its allies" suggests that the geopolitical entity mentions "United States" and "allies" are `Targets` of the event, but the annotation guidelines forbid to annotate them as such because geopolitical entities cannot be `Targets`, only persons, organizations, vehicles, facilities, and weapons can. Nevertheless, it seems incomplete to just discard "United States" and "allies" as arguments. There are two possibilities to overcome this: Either the constraints in the guidelines are to be rescinded, or the entity types of "United States" and "allies" are to be changed to *per* (person) because in this context the two mentions mentonymically stand for "the people living in the US and its allied countries".

2.2.6 ACE Documents

ACE offers 599 annotated documents, grouped into six genres: Usenet newsgroups (un; newsgroups), broadcast conversations (bc; conversations), telephone conversation transcripts (cts; transcripts), weblogs (wl), broadcast news (bn; broadcasts), and

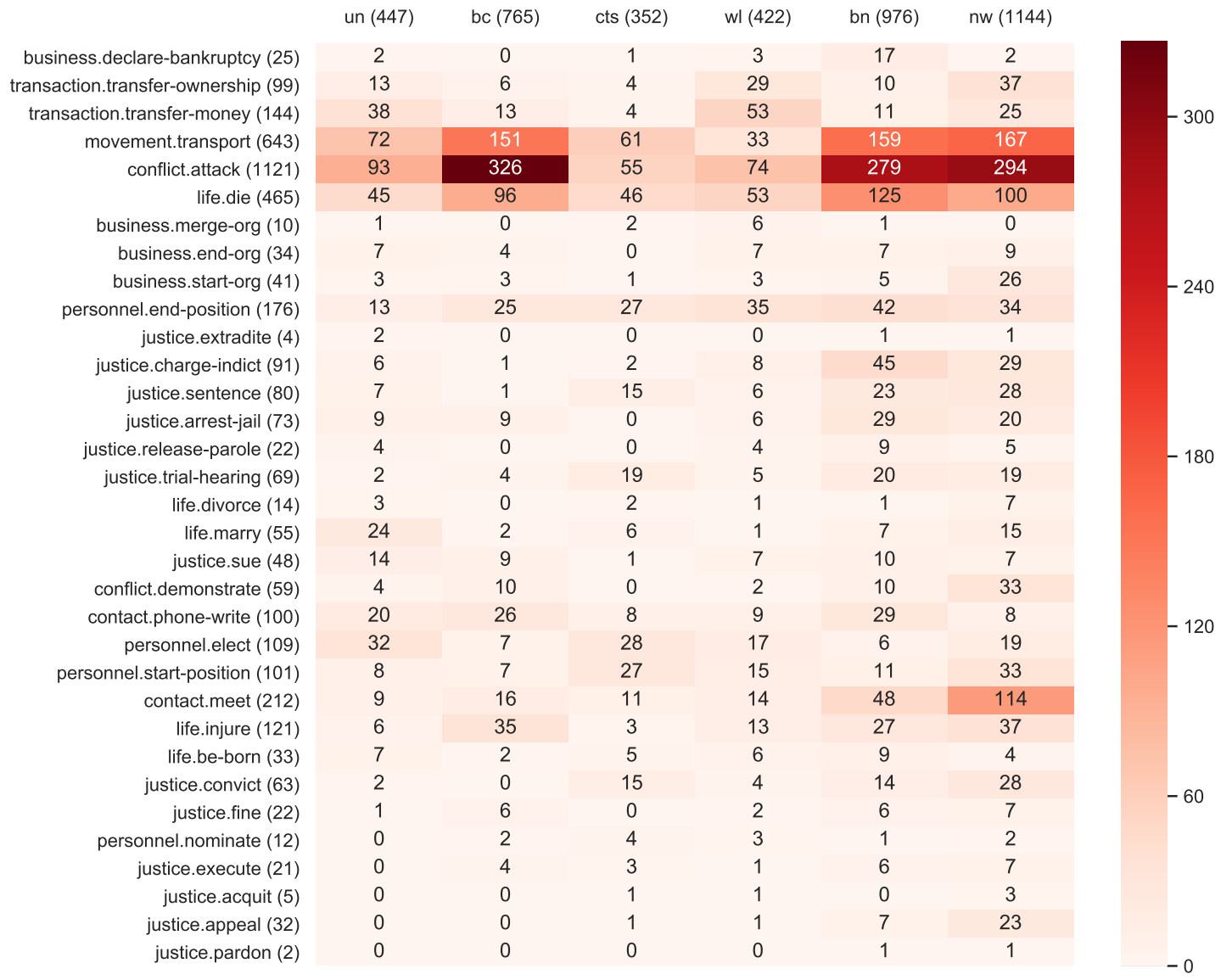


Figure 2.1: A heat map (darker colors mean higher values) representation of event types (y axis) per ACE genre (x axis) distribution. Genres are: Usenet newsgroups (un), broadcast conversations (bc), telephone conversation transcripts (cts), weblogs (wl), broadcast news (bn), and newswire (nw). Numbers in parentheses are the sums of the respective row or column values.

newswire (nw). Figure 2.1 depicts the distribution of events to genres. Event types are on the y axis and genres on the x axis. In parentheses, we report the total number of instances per row (of event types in total) and column (of event types in the respective genre). For example, there are 25 DECLARE-BANKRUPTCY events in ACE, 2 in newsgroups, 1 in transcripts, 3 in weblogs, 17 in broadcasts, and 2 in newswire. Newsgroups has 447 events, broadcasts 762, etc.

In total, we have 4106 event annotations – a low number for a complex task like event extraction. Some of the events are infrequent. DECLARE-BANKRUPTCY has only 25 instances, MERGE-ORG 10, and EXTRADITE 4. ATTACK is the most frequent, PARDON the most infrequent event type (1121 and 2 annotations, respectively). ATTACK is the most frequent event type in five of the six genres; the only exception is transcripts, which contains more TRANSPORT events. ATTACK, TRANSPORT, and DIE clearly dominate the other event types in terms of frequency.

Conversations, broadcasts, and newswire have similar event distributions – with the notable exception of a considerably higher amount of MEET events in newswire. Newsgroups, transcripts, and weblogs are also similar to each other, with the exception of considerably less TRANSPORT events in weblogs.

In Chapter 5, we directly address the low amount of training data by introducing bagging as a training regime to event extraction (Section 5.6.3). Scarce training data also makes indeterministic training effects more severe (Section 2.5).

Most publications use the train-dev-test split introduced by Ji and Grishman (2008). This split uses 30 documents as development data, and 40 documents as test data. Note that the test set consists only of newswire articles and ignores all other genres. We introduce two additional data splits which follow ACE's genre distribution more closely in Chapter 5.

2.3 ACE Events in a Bigger Context

In ACE, event extraction is an information extraction task, and events are entities with a trigger word, a set of roles, and arguments. Furthermore, ACE documents only cover news and political discussions. In this section, we want to outline event tasks from other domains (Section 2.3.1) as well as other event formalisms (Section 2.3.2). Finally, we describe tasks which are similar to event extraction (Section 2.3.3).

2.3.1 Other Domains

The BioNLP'09 event shared task (Kim et al., 2009) is prominent in the bio(medical) NLP community. The task resembles ACE event extraction in many aspects. Both assume gold entities are given, and both require the identification and classification of triggers and arguments. Furthermore, both schemes encode event modifications like negations or speculations. However, the domains differ. In the BioNLP shared task, events are concerned with protein biology; triggers express biological processes (e.g., gene expression), arguments are proteins and other biological events. The last point (other events as arguments) is a fundamental difference between the BioNLP shared task and ACE (including subsequent annotation schemes like ERE): ACE explicitly neglects event-event interactions.

2.3.2 Other Event Formalisms

TimeML (Pustejovsky et al., 2003a) is an annotation formalism which highlights the temporal aspects of events. It was developed for question answering, especially for questions involving temporal expressions (e.g., 'currently') or asking about points in time ('when did'). TimeML addresses four temporal event problems: (a) identify an event and anchor it in time, (b) order events either based on their absolute temporal order or based on their 'lexical' ordering in a discourse, (c) reason about underspecified temporal expressions (e.g., 'last week'), and finally (d) reason about the duration of an event. It also includes event-event relations: Temporal relations (before/after), subordinations (e.g., if one event provides evidence for another: 'he said he did'), and aspectual relations (e.g., if one event initiates the other: 'he started to read').

TimeML was used to produce the TIMEBANK corpus (Pustejovsky et al., 2003b), which provides several thousand events and information about events as described above. It also served as the foundation of SemEval 2007 Task 15 (Verhagen et al., 2007), a shared task about identification of temporal relations, and its successor, SemEval 2010 Task 13 (Verhagen et al., 2010).

2.3.3 Structurally Similar Tasks

In this section, we want to discuss two semantic formalisms which are structurally identical to event extraction (both have equivalents for triggers and arguments). Both encode predicate-argument structures, a more general concept than events. From a

computational point of view, the three tasks are equivalent. At least in theory, they can use the same infrastructure and inference procedures.[6]

Semantic Role Labeling, or SRL (Gildea and Jurafsky, 2002; Màrquez et al., 2008), is an NLP task and a semantic formalism. SRL is usually introduced as a formalism describing events. However, this notion of 'event' is different from the one used in event extraction. In the latter, an event is *something important that happens* – in the former, an event is *something that happens*. In other words, SRL is concerned with characterizing all events which occur in natural language, whereas event extraction seeks for the extraction of an exclusive list of pre-defined events of interest.

SRL aims to bridge syntax and semantics in a 'useful' way (Palmer et al., 2005) by mapping semantic relationships onto predicate-argument structures, e.g. by indicating the agent among all arguments of a predicate. The predicate (typically a verb) determines the event. The event schema also serves as a disambiguation schema – it encodes polysemy for example. As in event extraction, different events have different role sets. The totality of the roles for some event is called a frameset. Framesets come in two versions. One consist of numbered roles: Arg0, Arg1, etc., where Arg0 is similar to a Prototypical Agent (Dowty, 1991) and Arg1 is similar to a Prototypical Patient or Prototypical Theme. The other consists of predicate-specific roles. However, SRL systems usually only predict general roles only.

Frame semantic parsing (FSP) is based on *frame semantics* (Fillmore, 1982) and FrameNet (Fillmore et al., 2003). Like SRL, FSP encodes semantic information in predicate-argument structures. Unlike SRL, there are no prototypical role sets. In this respect, FSP is more similar to event extraction. The task is to predict frames ('event types') for lexical units (trigger equivalents) and their frame elements (argument equivalents). Lexical units are mostly nouns or verbs. Other parts-of-speech also include adjectives and prepositions. Frame elements are frame-specific, much like the event-specific role sets in event extraction, or the predicate-specific framesets in Semantic Role Labeling.

Judea and Strube (2015) retrain SEMAFOR, a well-known frame-semantic parser (Das et al., 2014), to extract events. With only a new frame element feature set, the retrained system can rival the then-state-of-the-art in event extraction on predicted entity mentions (Li et al., 2014). Judea and Strube (2015) identify two major problems in re-training SEMAFOR for event extraction, which can be interpreted as two major differences between event extraction and frame-semantic parsing. First, in frame-

[6] If the system is feature-based however, each task would need its own feature set.

semantic parsing, there is no 'negative class' – each lexical unit triggers some frame, whereas in event extraction, many eligible trigger candidates are in fact not a trigger for any of the pre-defined event types. Second, ACE event arguments are always entity mentions, whereas frame elements often correspond to syntactically well defined structures like noun phrases, which makes it easier to identify frame elements because they can be deduced from syntax parser output.

2.4 ACE Event Extraction as a Computational Task

In this section, we describe the computational task 'event extraction'. Afterwards, we define technical terms regarding the decoding process of event extractors. In the next section, we will look at event extraction as a scientific pursuit. In the following, we have some information overlap with previous sections. Here, we are only interested to introduce event extraction as a computational task and to describe the implications and structures we face.

There are different names for the actual task: ACE speaks of 'Event Detection and Recognition' (ACE2005, 2005). Some of the publications in the field have only 'event detection' in their titles if they only predict event triggers (Feng et al., 2016; Liu et al., 2017, i.a.). However, the task is most frequently referred to as 'Event Extraction'.

Event extraction consists of two sub-tasks, corresponding to the two main structures of ACE events, namely detecting and classifying triggers and arguments. Detection refers to finding words and entity mentions which are triggers and arguments of some event. Classification also involves to predict the respective trigger and argument types. Most publications report evaluation numbers for both tasks. In this thesis, we will continue this mode. Please note that there is also an increasing number of publications for trigger-only systems (Section 6.2).

ACE events (or rather event mentions) are defined intra-sententially: The trigger and all arguments of an event can always be found in the same sentence. Therefore, most event extractors operate within sentences. Finding event mentions is usually cast as a two-stage approach: A system first classifies each word in the sentence as belonging to one of the 33 ACE event types or a negative class. If an event type is predicted, the system looks for an argument type given the trigger and all entity mentions in the sentence, again including a negative class. However, the task can also

be modeled jointly. In Section 2.4, we formalize the different ways an event extractor can approach the task.

In terms of evaluation, most publications follow the procedures introduced in Ji and Grishman (2008): Triggers are correctly identified if their span matches that of any gold trigger. They are correctly classified if the event type is also correct. Similarly, arguments are correctly identified if their span matches that of any gold argument and they are correctly classified if the respective argument type/the role is correct. We follow most publications and usually report both, 'identification' and 'classification' scores for triggers and arguments.

In most publications, entity mentions are given, meaning that the extractors rely on gold entity mentions. There are only a few systems which predict entity mentions (Li et al., 2016, and our work in Section 3.3.2).

One of the main findings of this thesis is strongly related to the computational task: Trigger prediction has a fundamental impact on argument prediction. This is intuitively clear: For every missed trigger, we also miss all its arguments, and for every spurious trigger we may introduce spurious arguments. In Sections 3.3 and 5.7 we show that it is trigger *recall* which bears the most impact, and not trigger prediction performance in general. It is especially true that a system can significantly increase its argument prediction performance solely by increasing trigger prediction recall.[7] This has some implications for comparability, especially if one claims that a system has a better argument prediction performance than another: The effect can be solely based on increased trigger recall. In Chapter 3 for example, we present a system which improves trigger prediction by global inference – this in turn substantially increases argument predictions as well. However, the system actively only improves trigger predictions. To the best of our knowledge, we are the first to explicitly mention and investigate this effect. In Chapter 4, we devise a setting where trigger predictions have no effect on argument predictions in order to reliably evaluate the impact of syntax encodings to argument predictions.

Terminological Clarifications

In the following, we introduce six technical terms (organized in three categories) which we frequently use in this thesis. Four of them describe the prediction process.

[7] Clearly, the positive effects of increased trigger recall diminish if the loss in precision is too severe.

We also use the distinctions we make here while introducing the terms to characterize the most influential event extractors published before 2018 (Section 6.1).

We first describe *decoding*, before we introduce the terms *joint vs. disjoint* and *local vs. global* to characterize how most event extractors model the decoding process. We also introduce and formalize the terms *static vs. dynamic* for feature templates.

'Decoding' refers to the process of creating and labeling event structures (triggers and their arguments); we use the term interchangeably with 'inference'. *Joint vs. disjoint* refers to the decoding type and *local vs. global* to the decoding scope.

Decoding type is a qualitative dimension – are triggers and arguments predicted jointly or consecutively? In *joint decoding*, predictions influence each other. Joint event decoders typically predict the event structure of an entire sentence. They do not settle for a definitive answer on any trigger or argument labeling until the entire sentence is labeled for triggers, and all entity mentions are assigned a label with respect to each trigger. *Disjoint decoding* on the other hand predicts triggers and arguments, or arguments among themselves, independently and consecutively. The decision for a trigger label is made *before* the argument labeling begins. In other words, the decisions do not influence each other. A special type of inference is given if future decisions are informed by previous ones, without the possibility to revoke the previous ones. We characterize this as disjoint decoding.

Decoding scope a quantitative dimension – how far does information flow within a document? *Local decoding* is only informed about the current sentence, or even only about smaller contexts. *Global decoding* crosses sentence boundaries and can draw information from the entire document, or even multiple documents.

Most event extractors can be characterized using the two dimensions above. To further clarify the distinctions, consider the following example.

(19) . . . demonstrating against military strikes on Iraq and calling on Muslims to wage jihad against the United States and its allies.

. . .

I don't think America will win this war, as our jihad and our resistance will teach the Americans and British a lesson they will never forget," he said.

Local and disjoint systems (which applies to most published event extractors) would first predict triggers and then arguments for each trigger. They may first predict that

"demonstrating" probably triggers a DEMONSTRATE event. We will assume argument prediction returns no arguments for this trigger. Then, local and disjoint systems may predict that "strikes" is an ATTACK trigger. Argument predictions follows. They may predict that "Iraq" is a `Target`. They may also predict that "United States" is a `Target` because they would not be informed that they already predicted a better `Target` – in this example, "Iraq" and "United States" look very similar to each other in terms of features.

Local and joint systems on the other hand operate on a joint search space, preferably encompassing all possible decisions for the entire sentence. Here, the presence of "Iraq" as a possible `Target` would also inform the decision for "strikes" and vice versa, and the presence of "strikes" as an ATTACK trigger would inform the decision for "demonstrating". Furthermore, a joint system would be able to determine that "Iraq" is a better `Target` than "United States" for this event, and may not assign the role twice. Joint systems have a clear advantage over disjoint ones because they are less prone to error propagation – false decisions can be revised as more information becomes available.

A special kind of joint decoding is pattern matching. For example, Grishman et al. (2005) collect training data patterns which characterize the connection of a trigger to all arguments, and apply pattern matching at test time to check if events are present in new sentences. If a pattern matches, the respective trigger and argument assignments constituting the pattern are believed to be present in the sentence as well. This means, that trigger and argument decisions are performed jointly without the need to search through a large joint decoding space.[8]

However, local systems (regardless of decoding type) would predict all occurrences of "jihad" as ATTACK triggers, because the word never appeared as such in the training data. Local systems might get the first occurrence right, but the second seems to be more difficult. Global systems are not limited to a sentence. They can draw information from all occurrences of "jihad" and inform all of them in turn. They can also harvest information from other events throughout a document. In Section 3.2, we present a joint and global system.

Handling a joint or global hypotheses space is demanding. Even for short sentences, searching the entire space is not possible without approximate methods like beam search. Furthermore, joint search spaces require dynamic features, i.e., features capturing interactions of classification decisions. We call the templates gener-

[8] However, Grishman et al. (2005) refine their argument decisions disjointly in subsequent steps.

ating such joint features dynamic because their actual values change over time and across hypotheses. Dynamic features either model trigger-argument, trigger-trigger, or argument-argument interactions (e.g., argument roles one entity fills in different events). We can note that joint and/or global systems need dynamic feature templates.

The ideal event extractor is joint and global because trigger and argument decisions influence each other and depend on other event decisions throughout the document. The ideal system uses the entire information in ACE documents to produce coherent event assignments. To the best of our knowledge, there are only two clearly joint and global event extractors, Yang and Mitchell (2016) and our work in Chapter 3.

2.5 ACE Event Extraction as a Scientific Pursuit: Comparability and Reliability

The standard evaluation setting for most event extraction papers is to report numbers for trigger and argument identification and classification. Most papers and this thesis adopt the evaluation scheme presented in Ji and Grishman (2008): A trigger is correctly identified, if its span matches any gold span; it is correctly classified, if its event span and event type match those of any gold trigger. An argument is correctly identified, if its span matches any gold span; it is correctly classified, if its span and argument type (role) match those of any gold argument. Both, identification and classification scores are usually reported.[9] Evaluation measures are always precision, recall, and F_1 (Manning and Schütze, 1999). We agree with and mostly follow established evaluation procedures. However, we want to address some problems which weaken the reliability of evaluation results, especially in the context of deep learning methods. We will present the problems and then propose a solution. In addition to the measures we take here, we also motivate and use new data splits in Chapter 5.

The common belief in NLP is that testing on the same test set enables comparability. The usual conclusion is that one system, and as an extension the underlying method, is better than another if it increases the same evaluation metric on the same test set, given that the evaluation measure and the test set represent the task adequately. However, we believe that it is not enough to just use the same test set in order to ensure comparability in a strict (scientific) sense, and to claim that method B (in contrast to

[9] Entire events are almost never evaluated in publications. The only exception we are aware of is Miwa et al. (2014).

system B) is better than method A. We believe that preprocessing and the indeterministic training procedures of deep learning systems have a great impact on results and weaken direct comparability between different systems. We raise the following two questions and discuss them in the following sections.

1. Do the systems use the same preprocessing? Better preprocessing might lead to better results, even for identical systems. We believe that it is difficult to claim the superiority of a *method* without identical preprocessing (Section 2.5.1).

2. Is training deterministic? Indeterministic training produces different models for the same data and hyperparameters, and these models might have significantly different evaluation numbers, even though they are instantiations of the same method, system, and preprocessing (Section 2.5.2).

2.5.1 Preprocessing

Preprocessing takes place before the actual input is presented to the system. It can involve manipulating (scaling, normalizing, etc.) or producing information (part-of-speech tagging, dependency parsing, word embeddings, etc.). Crone et al. (2006) investigate preprocessing effects for multiple classifiers (Decision Trees, Support Vector Machines, and Neural Networks) on different data mining problems. They find that the preprocessing (which was only of the manipulating kind in their case) has a significant impact on all methods and parametrizations. They also find that performances are as sensitive to preprocessing as they are to hyperparameters. Reimers and Gurevych (2017) report that the choice of word embeddings (another preprocessing input) has a great influence on six NLP tasks, across a wide range of hyperparameters.

We draw the conclusion that our systems/system versions have to use the same preprocessing if we directly compare their results. This especially includes comparisons to baselines. For example, we carry out a comparison of different syntax encoders in Chapter 5, and we use the exact same preprocessing and system to support them, and to enable direct comparability.

2.5.2 Indeterministic Training

Indeterministic training became common with the renaissance of neural networks. Neural network weights are typically initialized by drawing from a normal or uniform

distribution (Glorot and Bengio, 2010; LeCun et al., 2012). Initial weights are different each time training is started. This is the main reason why training neural networks is an inherently indeterministic process: It produces a different output (weights) for the same input (training data and hyperparameters). We quantify this effect in Chapter 5 where we compare different syntax encoders.

Note that one can force neural network training to be deterministic in the sense above (same input = same output) by fixing random seeds. However, this does not solve the problem of indeterminism because the output now depends on the exact random seed and the method which was used to produce pseudorandom numbers. Additionally, disabling randomness is not possible when training on GPUs because atomic GPU operations are asynchronous by design.[10]

Reimers and Gurevych (2017) and Reimers and Gurevych (2018) investigate this point thoroughly for a variety of NLP tasks, preprocessings, and hyperparameters. For example, they find that on the same data split identical Named Entity Recognition systems produce significantly different evaluation results 26% of the time, solely because their weights are initialized randomly.

ACE trigger classification is even worse – Reimers and Gurevych (2018) report that 34.5% of the time identical systems produce significantly different evaluation results, with fluctuations up to 9 test set F_1 points (4.3 F_1 points difference in the 95% percentile). In Chapters 4 and 5, where we also use deep learning methods, we observe fluctuations of up to 2 F_1 points, even though we use a variety of methods to reduce weight fluctuations during training. Please note that the typical new state-of-the-art improvement in papers which report ACE 2005 evaluations is around 1-2 F_1 points for trigger and argument classification, meaning that we can surpass state-of-the-art results by sheer luck.

We can conclude that there is a very high risk in reporting only one test set evaluation when using non-deterministic training procedures – the numbers may very well be due to chance. As Reimers and Gurevych (2017), Reimers and Gurevych (2018), and Chapters 4 and 5 show, even identical deep learning systems produce very different evaluation results when trained and evaluated multiple times. This hinders reproducibility and enables false conclusions about the superiority of a system or method.

To overcome this problem, we follow Peters et al. (2018) and always report evaluation numbers averaged over five models (and consequently five evaluation runs); we

[10] This is true for NVIDIA GPUs and the predominant CUDA framework, especially when using cuDNN, as of September 2018.

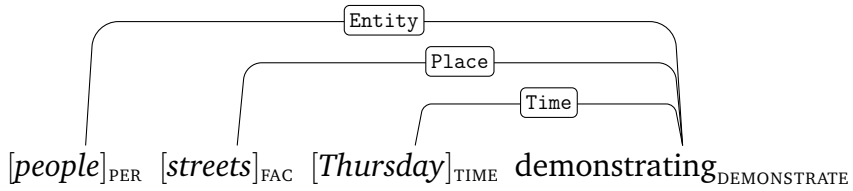
also report sample standard deviations for almost all evaluation numbers. Chapter 5 also reports evaluation of two additional, randomly drawn data splits to increase the reliability of our comparisons there.

We will not present the three main chapters of this thesis. We start with Chapter 3 which proposes a new inference method to make the global context of a document available to the decoding process of a local system. We use it with a joint base system to reach new state-of-the-art results in ACE event extraction.

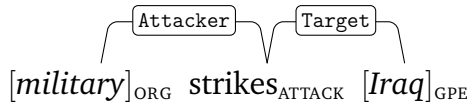
3 Global Event Extraction

> Tens of thousands of [*people*]$_{\text{PER}}$ took to the [*streets*]$_{\text{LOC}}$ across the [*Middle East*]$_{\text{GPE}}$ [*Thursday*]$_{\text{TIME}}$, demonstrating against [*military*]$_{\text{ORG}}$ strikes on [*Iraq*]$_{\text{GPE}}$ and calling on [*Muslims*]$_{\text{PER}}$ to wage jihad against the [*United States*]$_{\text{GPE}}$ and [*its*]$_{\text{GPE}}$ [*allies*]$_{\text{GPE}}$.

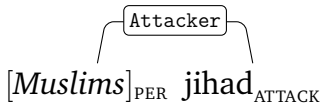
A sentence and its entity mentions are the starting point for most ACE event extractors. The task is to predict which tokens trigger events of interest, and which entity mentions play roles in them. We can find three events in the sentence above, triggered by the words "demonstrating", "strikes", and "jihad". We depict them below.



(a) First



(b) Second



(c) Third

Figure 3.1: All three event structures occurring in the introductory sentence.

The first is a DEMONSTRATE event (Figure 3.1a) triggered by "demonstrating". It has three arguments: "people" filling the `Entity` role, "streets" filling the `Place` role, and "Thursday" filling the `Time` role. The other events are ATTACKs triggered by "strikes" (Figure 3.1b) and "jihad" (Figure 3.1c), with "Iraq" as the `Target` and "military" as the `Attacker` of the first event, and "Muslims" as the `Attacker` of the second.[1]

[1]The event triggered by "jihad" seems to be incomplete because the sentence suggests that "United States" and "allies" should both be `Targets`. See the discussion to Example (18) in Section 2.2.5 for further details. From world knowledge and other parts of the same article, we can also infer that "United States" and the "allies" are `Attacker` of the first ATTACK event triggered by "strikes".

ACE events are exclusively intra-sentential in their scope. However, a single sentence often lacks important information. "jihad" for example occurs only twice as an event trigger in the entire training data, and the above sentence alone does not give strong clues that the word triggers an ATTACK event. This lack of information makes it hard for an intra-sentential event extractor to make a correct prediction.

Missing triggers do not only negatively impact trigger performance, they also interfere with argument classification, because all arguments of an unrecognized trigger become unaccessible for prediction. Especially *missing* triggers have a negative impact on argument predictions (Sections 3.3 and 5.7.3).

In this chapter, we present a system which casts event extraction as a structured prediction problem and employs global, document-wide inference to increase the overall performance of a local and joint predictor. The new system finds considerably more events in the sentences and increases both, trigger and argument performance. With global inference, "jihad" is not an isolated word anymore; it has semantic relations to other words in the document like "war" which help to correctly recognize it as an ATTACK trigger. Our contributions in this chapter are the following.

1. We present an efficient global inference method which enables a joint and local base system to access information from the entire document (Section 3.2). This method is in agnostic to the actual base system. We use it with two related but different base systems, one which uses gold, and one which uses predicted entity mentions. The evaluation in Section 3.3.1 shows that our system outperforms even the most recent deep learning methods .

2. We introduce a new feature set to event extraction (Section 3.3.1). Some are specifically designed for our global decoding method.

In Section 3.1 we present our main base system – a state-of-the-art event extractor which predicts event triggers and arguments jointly and locally. In Section 3.2 we argue that a system which operates on individual sentences is often not enough. We present a method to access the global (document-wide) context, and we present a set of new features to guide global inference. Section 3.3 reports experiments and results. Finally, Sections 3.4 and 3.5 discuss and conclude our findings, respectively.

3.1 Intra-Sentential Event Detection

In this section we describe and analyze our base system (Li et al., 2013), a local and joint event extractor.[2] Section 3.1.3 describes features which are only possible in the joint setting and cannot be used by disjoint systems. Among other things, these features model the interaction between different event types within a sentence. With these features, joint systems have a theoretical advantage over any disjoint system. However, more recent, disjoint deep learning systems (Chen et al., 2015; Nguyen et al., 2016, i.a.) outperformed our base system, presumably because of word embeddings and a better modeling of lexical contexts. In this thesis, we also come back to disjoint, deep learning methods, mainly to explore the use of syntax encoders for event extraction.

In Section 3.2 we show that joint event extraction is not sufficient, because it is still limited to the actual sentence, which often lacks sufficient information to find all events. In Section 3.2 we present a global decoding method which enables the base system to use information from the entire document. With global decoding, our system outperforms most recent event extractors without using word embeddings or deep learning techniques. We will now describe the base system.

3.1.1 Decoding and Training

In the following, we describe our base system. This system is identical to Li et al. (2013). However, we deviate from their description and formalize the system differently. We also provide additional information to the base system which is not reported in their paper, but appears in their code.

Given a sentence s, our goal is to predict all event triggers and arguments for s. One possible assignment containing all triggers and arguments of s is called a *configuration*. Our problem can be rephrased: We want to find the best configuration $\hat{c}$ given s:

$$\hat{c} = \underset{c \in C(s)}{\arg\max}\ f(s, c) \cdot w. \tag{3.1}$$

$C(x)$ is a function which enumerates configurations for s, $f(s, c)$ is a feature function, and w a weight vector. Finding $\hat{c}$ is a *structured prediction problem*: We search a

[2] 'Joint' and 'global' are defined in Section 2.4.

large space for the best structure $\hat{c}$. We aim to solve this problem using a *structured perceptron* (Collins, 2002; Huang et al., 2012).

An exact solution of Equation 3.1 is prohibitive. We cannot enumerate all possible configurations. Given the 33 ACE event types, we already have 33^m configurations for a sentence of length m for trigger assignments alone. With argument assignments, the number is much higher.

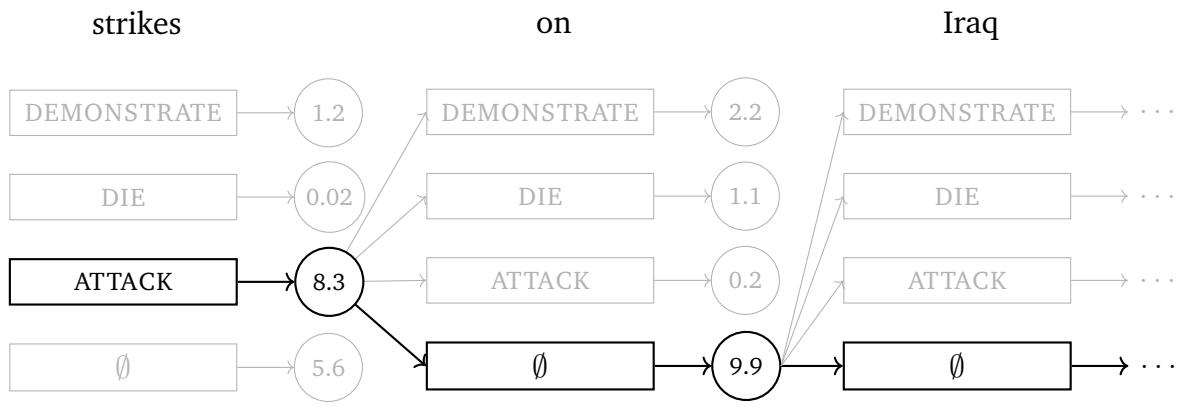
To mitigate this problem. we follow Li et al. (2013) in using approximate search for decoding, more precisely *beam search*. Instead of having $C(x)$ enumerate entire event structures for a sentence, we construct configurations iteratively and prune the search space after every iteration. In the following, we describe this process. Afterwards, we describe the feature function $f(s, c)$ and how to learn a good feature vector w.

Configurations are built word-by-word using two actions: *trigger assignment* and *argument assignment*. The first action assigns labels to words (either an event type or $\emptyset$), the second action assigns labels to entity mentions (either an argument type or $\emptyset$). In the settings where we avoid entity prediction (the predominant setting in ACE event extraction), the configurations already contain all entity mentions. Figure 3.2 visualizes a snapshot of a decoding pass. We describe the process below and refer to the figure where necessary.

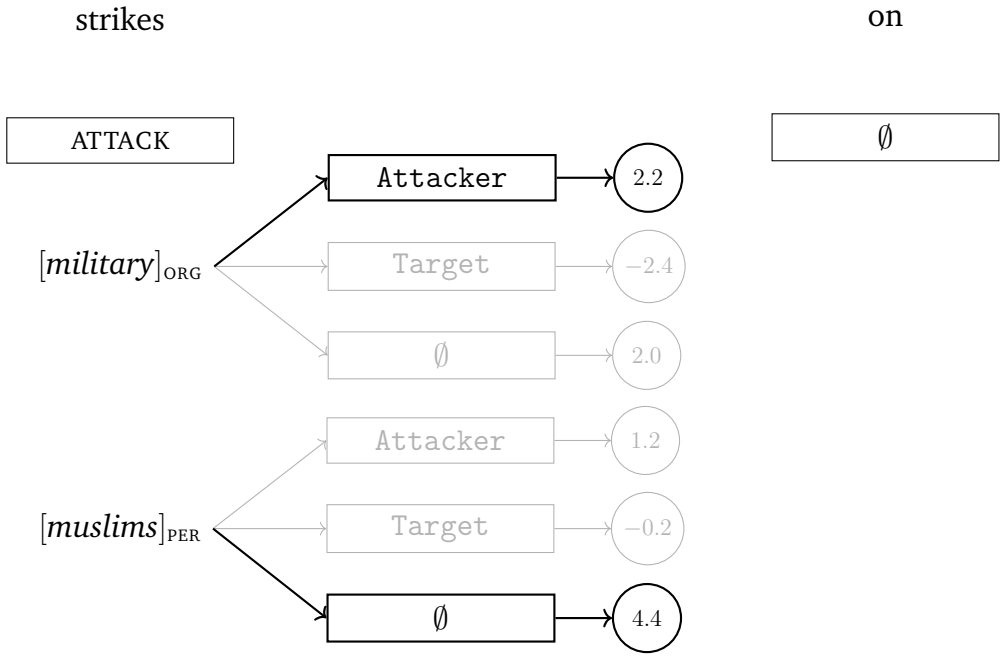
Decoding starts with the first word[3] in the sentence, "strikes" in our example. For the sake of simplicity, we ignore all previous words from our descriptions and visualizations. TRIGGER ASSIGNMENT generates all possible labels for "strikes", 34 in total (33 event types plus $\emptyset$). In Figure 3.2, this is exemplified by the four event types below the word. Each such assignment constitutes a new configuration which contains all previous assignments plus the new one. "Strikes" as a DEMONSTRATE or as an `Die` trigger are two assignments for example, and two different configurations. The new configurations are scored and only the top n are retained. In Figure 3.2, $n = 1$, so we only keep the best configuration [strikes : ATTACK].

After new trigger assignments are generated, ARGUMENT ASSIGNMENT is executed for each event trigger. In our case, we assign only one event type, namely ATTACK, to the word "strikes". Now, we have to find arguments of this event. The procedure assigns one of 29 labels (28 argument types plus $\emptyset$) to each entity mention in the sentence. Each argument assignment again results in a new configuration. New configurations are scored and only the top n retained. This process is visualized in Figure 3.2b, exemplified for the case that "strikes" is an ATTACK trigger. The procedure lists

[3] In this thesis, we use "word" and "token" interchangeably.



(a) Illustrates TRIGGER ASSIGNMENT with three words, four possible labels, and beamsize 1. Words are processed consecutively. At each word, all possible trigger labels (event types) are enumerated, and each of them paired with all previously computed configurations. Each such pairing constitutes a new configuration. After each word, only the best configuration (beamsize 1) is retained. Assignments in bold are top (best-scoring) assignments, the bold path constitutes the best (and correct) configuration at each position. Numbers in circles are assignment scores, accumulated over time.



(b) Illustrates ARGUMENT ASSIGNMENT. For each event type in a configuration, argument label assignment is triggered. Depicted is the case that "strikes" is an ATTACK trigger and there are two entity mentions in the sentence. The system enumerates all roles for the event type (`Attacker`, `Target`, ∅) and assigns the highest-scoring role to the respective entity mention.

Figure 3.2: Visualization of a hypothetical decoding pass.

Algorithm 1: TRIGGER ASSIGNMENT(beam b, sentence s, token index i)

```
   /* beam buffer                                                    */
1  b_i ← ∅;
2  for c ∈ b(i − 1) do
       /* T_e is the trigger label set (33 event types and ∅)       */
3      for t_e ∈ T_e do
           /* test if POS of word i is allowed, and if i overlaps with an
              entity mention                                         */
4          if IS ASSIGNABLE(c, i, t_e) then
               /* create new trigger assignment for position i      */
5              n ← [i : t_e];
               /* extract features                                  */
6              EXTRACT FEATURES(n, s);
               /* copy configuration c and add new trigger assignment */
7              b_i ∪ (COPY(c) ∪ n);
8  return b_i;
```

all allowed argument types for an ATTACK event (in pour example, `Attacker`, `Target`, and $\emptyset$), scores them and keeps only the n highest scoring assignments (in our example, $n = 1$). This is executed for both entity mentions ($[\textit{military}]_{\text{ORG}}$ and $[\textit{muslims}]_{\text{PER}}$).

TRIGGER ASSIGNMENT and ARGUMENT ASSIGNMENT enumerate partial event structures (trigger and argument labels, respectively), with each new assignment resulting in a new configuration which is then scored and discarded if it is not in the top n. This is exactly the approximate beam search we use; instead of enumerating millions of configurations for a sentence, we always restrict the search space to n hypotheses. The lower n is, the more greedy and faster beam search becomes.

After we enumerate and score trigger and argument assignments for the first word, the procedure moves to the second word, "on". Again, TRIGGER ASSIGNMENT generates all possible labels. Configurations without argument assignments constructed so far include: [strikes : ATTACK, on : DEMONSTRATE], [strikes : ATTACK, on : DIE], etc. Again, only the top n configurations (in our case, [strikes : ATTACK, on : $\emptyset$]) are kept for further processing and ARGUMENT ASSIGNMENT takes place for each new trigger. Since "on" is never a trigger in any new configuration, this step is omitted at this point in the example.

After the last word in the sentence is decoded, n configurations remain, each with event and argument label assignments for each word and entity mention in the sen-

Algorithm 2: ARGUMENT ASSIGNMENT(beam b, sentence s, trigger index i, entity mention index j)

```
   /* beam buffer                                                    */
1  b_i ← ∅;
2  for c ∈ b(i) do
3      b_i ∪ COPY(c);
4      if IS TRIGGER(c(i)) then
           /* T_a is the argument label set (28 argument types and ∅)  */
5          for t_a ∈ T_a do
               /* test if trigger type c(i), entity type of mention j, and
                  role t_a are compatible                             */
6              if IS ASSIGNABLE(c, i, j, t_a) then
                   /* create new argument assignment for position i and
                      entity mention j                                */
7                  a ← [j, i : t_a];
                   /* extract features                                */
8                  EXTRACT FEATURES(a, s);
                   /* copy current configuration c and add new argument
                      assignment                                      */
9                  b_i ∪ (COPY(c) ∪ a);
10 return b_i;
```

tence. TRIGGER ASSIGNMENT is visualized in Figure 3.2a and formalized in Algorithm 1. ARGUMENT ASSIGNMENT is visualized in Figure 3.2b and formalized in Algorithm 2.

Following Li et al. (2013), we use the structured perceptron as a learning algorithm. The perceptron is one of the oldest learning algorithms. It uses a hand-crafted set of categorical features for which it learns weight – one per feature. Our perceptron is 'structured' because it learns to classify more complex structures than the usual, unstructured classification samples; in our case, it learns to score correct configurations higher than incorrect ones. Algorithm 3 formalizes the structured perceptron with beam search, both during training and testing.

Input to Algorithm 3 is a sentence s, a gold configuration y, the beam size n and a flag indicating training mode. The algorithm initializes an empty beam b. A beam is a ranked list of (partial) configurations. Beams are indexable – $b(i)$ returns configurations containing trigger and argument assignments up to the word at index i.

After the beam initialization, the algorithm iterates through word positions $1 \dots |s|$ and calls TRIGGER ASSIGNMENT (Line 3). Each time it assigns an event type, it iterates through all entity mentions $1 \dots |\textit{mentions}|$ and calls ARGUMENT ASSIGNMENT for each (Line 10). New configurations are scored and only the top n configurations are retained (Lines 4 and 5 for triggers, and 11 and 12 for arguments).

Lines 7, 14, and 17 are important for learning: As soon as the partial gold configuration y_i is not predictable because the beam does not contain it anymore, we update feature weights and exit beam search. This is called *early update* (Collins, 2002; Huang et al., 2012). Huang et al. (2012) show that updating only after decoding completed leads to bad performance with inexact search strategies such as beam search. If the model is not penalized as soon as the gold configuration is not predictable anymore, we might perform updates which actually lower the score of the gold configuration – such updates are called *invalid updates*. If we update immediately when the gold configuration becomes unpredictable we drive the model towards predicting an increasing portion of the correct configuration. If we update only after decoding is finished, we might actually penalize parts of the model which would have lead to the correct solution if previous errors would not have been made. Huang et al. (2012) show under which conditions the structured perceptron converges. Their most important finding is that the proof does not require exact inference; approximate inference like beam search suffices, as long as the updates are guaranteed to contain violations. If the correct solution is not part of the beam anymore, we cannot guarantee that the current best (and yet wrong) partial solution would be ranked higher than the correct partial

Algorithm 3: BEAM SEARCH(sentence s, gold configuration y, beamsize n, training)

beam $b \leftarrow \emptyset$;
for $i = 1 \ldots |s|$ **do**
    /* make new configurations with new trigger assignments, compute feautures */
    $b(i) \leftarrow$ NODE ASSIGNMENT(b, s, i);
    /* prune the beam */
    SCORE AND SORT$(b(i))$;
    $b(i) \leftarrow$ TOP$_n(b(i))$;
    /* update weights if we are in training mode and the gold label y_i fell out of the beam */
    if *training* $\wedge\ y_i \notin b(i)$ **then**
        UPDATE$(b(i, 1), y_i)$;
        exit;
    for $j = 1 \ldots |$*mentions*$|$ **do**
        $b(i) \leftarrow$ ARGUMENT ASSIGNMENT(b, i, j);
        /* prune the beam */
        SCORE AND SORT$(b(i))$;
        $b(i) \leftarrow$ TOP$_n(b(i))$;
        /* if we are in training and the gold argument y_i^j fell out of the beam, update weights */
        if *training* $\wedge\ y_i^j \notin b(i)$ **then**
            UPDATE$(b(i, 1), y_i)$;
            exit;
/* update weights if we are in training mode and the top configuration assigned to the last word, $b(|s|, 1)$, is not equal to the gold configuration y */
if *training* $\wedge\ y \neq b(|s|, 1)$ **then**
    UPDATE$(b(|s|, 1), y)$;
    exit;
return $b(|s|, 1)$;

position; in other words, we cannot guarantee that the update fixes a violation. This is the reason why early updates almost alway lead to better performance compared to 'late updates' – early updates are guaranteed to contain a violation and therefore always push the model towards the correct solution.

Note that TRIGGER ASSIGNMENT and ARGUMENT ASSIGNMENT use IS ASSIGNABLE to test if trigger and argument label restrictions are met. For trigger labels, the function tests if the respective word has the correct part-of-speech, and if it does not overlap with entity mentions. Both factors are not mentioned in Li et al. (2013) but used in their code. A word can only receive a trigger label if its part-of-speech matches the regular expression in Appendix A.1. Mainly verbs, nouns (including proper nouns), and adjectives can get a trigger label. Furthermore, if a word overlaps with an entity mention, it cannot be a trigger candidate. The second test excludes event-specific attributes like mentions of crimes ("the crimes he's been convicted of"; "siphoning millions of dollars from Project Coast to finance a lavish, globe-trotting lifestyle and of selling drugs") because they can span big parts of a sentence and usually contain one or more triggers. For arguments the function tests if the combination of entity, event, and argument types are compatible; for example, *time*, ATTACK, and `Time` are compatible, but *time*, ATTACK, and `Target` are not, because *time* mentions cannot fill the `Target` role. If a combination is not possible, the respective argument type is omitted from ARGUMENT ASSIGNMENT.

If BEAM SEARCH is not in training mode, or if it did not make any prediction errors, it returns the top hypothesis ending at the last token position. For a sentence with m tokens this would be $b(m, 1)$.

3.1.2 Learning Weights

After each call of TRIGGER ASSIGNMENT or ARGUMENT ASSIGNMENT we check if the partial gold configuration y_i is still in the beam. If so, we continue processing. If not, we have to update feature weights and continue with the next training instance (UPDATE in Algorithm 3).

For each training instance (s^n, y^n), where s^n is the sentence and y^n the gold configuration of sample n, we compute the best configuration $\hat{c}_n$ according to Equation 3.1, implemented in Algorithm 3. If we have to update because y^n fell out of the beam, or because $\hat{c}^n$ is not the correct solution (Lines 7, 14, and 17 in Algorithm 3), we perform perceptron updates:

$$w^t = w^{t-1} + \alpha f(s^n, y_i^n) - \alpha f(s^n, c_i^n). \tag{3.2}$$

w^t is the new feature vector after, w^{t-1} the feature vector before weight updates. α is the learning rate (we set $\alpha = 1$), and f the feature function. f returns the features for the gold and the top predicted configurations up to position i (y_i^n and c_i^n, respectively).

Features produced in the gold configuration y_i^n are updated with a positive value, features produced for the predicted configuration c_i^n are updated with a negative value. Features common to both are not updated. After weight updates were performed, we continue with the next training sample. Note that it is imperative to randomize the order of training samples to reduce overfitting and to increase generalization.

When we apply the system to test data, we employ feature averaging (Collins, 2002): Instead of using w^t, we average over all feature vectors encountered during training. This has the effect that weights with high oscillation are smoothed. It has been shown that averaging yields considerably better performance for perceptrons. Weight averaging can also benefit probabilistic learning (Section 4.3.8).

3.1.3 Features

Our base system uses two types of feature templates. *Static* templates apply exclusively to one trigger or argument assignment. Features produced by these templates capture decoding-invariant properties like words, lemmas, part-of-speech tags, etc. *Dynamic* templates on the other hand produce features which capture decoding-variant properties like the event type of a previous trigger, or the roles an entity mention plays across events within a sentence. To put it in other words: Static features do not vary over time, in contrast to dynamic features.[4] Tables A.1 and A.2 in the appendix report all features of the base system.

3.2 Incremental Global Inference

Event extraction decoders can be classified according to their *type* and *scope*. Decoding type refers to *joint vs. disjoint* decoding – are triggers and arguments predicted jointly, or iteratively? Decoding scope on the other hand refers to *local vs. global* decoding

[4] Li et al. (2013) and Li et al. (2014) use the terms "local" and "global" instead of "static" and "dynamic"; however, we have to reserve the term "global" for our cross-sentence inference method described in Section 3.2.

– does the extractor predict events only intra-sententially, or does it cross sentence borders? Section 2.4 discusses these terms in more detail.

Our base system uses joint and local decoding: It predicts triggers and arguments jointly, but does not cross sentence boundaries. Local decoding is the most obvious limitation here. In some cases, local decoding may be sufficient. There are words (e.g., 'attack') which strongly indicate the presence of event types in news texts without the need for more context. However, there are many cases which cannot be inferred correctly from the information within a sentence. Consider the following example, which is an extension of our introductory sentence, and which we already used in Section 2.4.

(20) . . . demonstrating against military strikes on Iraq and calling on Muslims to wage jihad against the United States and its allies.

. . .

I don't think America will win this war, as our jihad and our resistance will teach the Americans and British a lesson they will never forget," he said.

Here, "jihad" is an ATTACK trigger and appears twice in the document. However, the word appeared only twice as an event trigger in the entire training data. Because the base system did not see the word often as a trigger during training, it has to rely heavily on context, more specifically on the local context. We can see that the first "jihad" is embedded in the phrase "wage jihad against", providing important clues for its prediction. Furthermore, there are demonstrate and attack triggers in the sentence, which is an additional (although weak) clue that "jihad" is an attack trigger. The base system is limited to this information. Only the global (document-wide) context provides additional clues.

Our assumption about event trigger distributions within one document is that the same word tends to trigger the same event type. This is similar to the one-sense-per-discourse assumption (Gale et al., 1992) and holds for 99.4% of the ACE triggers (Liao and Grishman, 2010). If we can classify the first "jihad" correctly, document-wide inference is able to infer that the second occurrence triggers the same event type.

There are also a few counterexamples. Consider the following sentence.

(21) That preemptive strike they put, I think somewhat elegantly, trying to strike at the head of the snake.

In example (21), there are two different parts-of-speech for the word “strike”. Furthermore, the second “strike” is used figuratively, also supported by its object: “strike at the head of the snake”. In contrast to the first occurrence, the second “strike” does not trigger a CONFLICT event. However, our method likely labels both as triggers. It seems very difficult to distinguish between such cases. It also seems not worth the effort because these cases are rather infrequent.

The second, less obvious limitation of our base system is that it can only consider what it already processed. This means that the local decoding is not only limited to sentences, but also to the current processing position. This limits the power of features capturing intra-sentential event-event interactions. One such feature for example is the bigram ‘last-event-type, current-event-type’. In our example above, this feature indicates that the system already decoded a DEMONSTRATE event when it attempts to decode the word “strikes”. However, the baseline system cannot have a reversed feature – indicating that “strike” is an ATTACK trigger when it decodes the word “demonstrating”.

In the following section, we clarify some terminology, before we present a method to make the entire document-wide context available to our baseline. Instead of one decoding pass per sentence, this method performs multiple passes per document and draws features from information made available by previous decoding passes.

3.2.1 Method

Our global decoding method *Incremental Global Inference*, or IGI (Judea and Strube, 2016) makes the entire document accessible for decoding. It consists of two interleaved parts: *global inference* and *global features*. The main idea is to increase trigger and argument classification performance by taking the classes of semantically related triggers and arguments into account. Instead of searching through a vast, document-wide hypotheses space, we refine classification for semantically related cases incrementally: We produce a first labeling of all trigger and argument candidates using the base system. Then, we use the event labels of semantically related words assigned in the previous pass to refine their event type classifications and start the next pass, again refining decisions, etc.

Global features are based on two principles. First, the one sense per discourse assumption – one word form tends to trigger the same event type throughout a document (Liao and Grishman, 2010). We extend this notion: Semantically and morphologically

related words tend to express related event types. Consider our example: "war" is an ATTACK trigger; its hyponym "jihad" (as a kind of war) also triggers the same event type. If we can capture such relationships between words distributed in a document, we can better infer the labels of words which occur in low-information contexts. Second, we use a feature set new to event extraction which captures a kind of semantic similarity between a word and the semantic field of an event type, represented by all trigger words for this type. This feature group enables Incremental Global Inference to better infer the labels of words never encountered during training.

We will now describe and then formalize Incremental Global Inference. Section 3.2.3 discusses other global inference methods reported in the NLP literature and identifies IGI as an *collective classification* algorithm.

First, IGI performs standard decoding as described in Section 3.1 for all sentences in the document. Then, an iterative process informs new passes about the decisions made in previous decoding passes. This information is used to refine decisions. The process is repeated a fixed number of times. During training, weights are updated only after the last pass was performed. Incremental Global Inference is depicted in Figure 3.3 and formalized in Algorithm 4.

Algorithm 4: INCREMENTAL GLOBAL INFERENCE(document D, beamsize n, global decoding passes m, training)

```
  /* r holds top configurations for the entire document; each sentence
     has none or exactly one top configuration, new top configurations
     overwrite older ones                                              */
1 r = ∅;
  /* collection                                                        */
2 for i = 1...m do
3     for sentence s ∈ D do
          /* no feature updates                                       */
4         r ∪ GLOBAL BEAM SEARCH(s, y^s, n, training = false, r);
  /* prediction                                                        */
5 for Sentence s ∈ D do
      /* feature updates if in training mode                          */
6     r ∪ GLOBAL BEAM SEARCH(s, y^s, n, training, r);
7 return r;
```

Input to Algorithm 4 is a document d, the beamsize n, the number of global passes m, and a flag indicating training mode. Output is a structure r holding all top con-

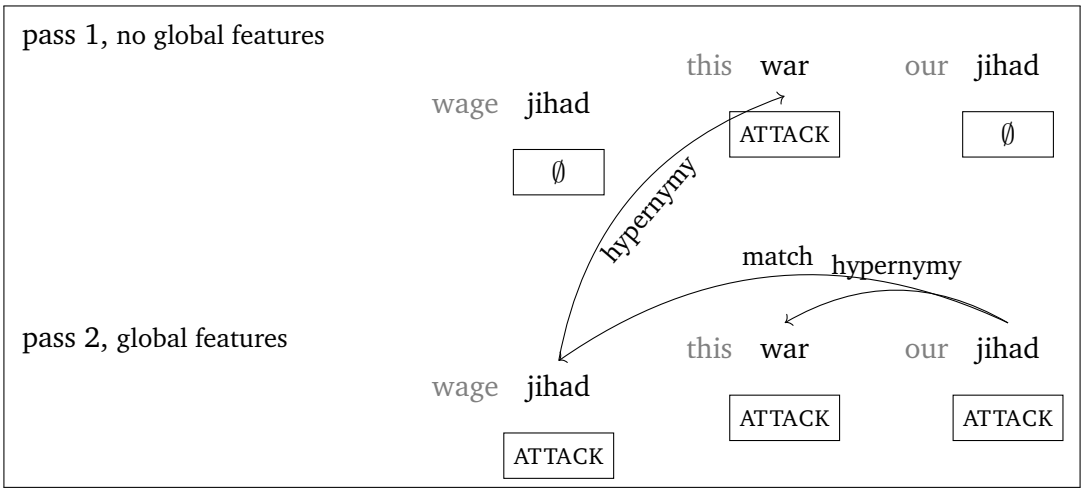


Figure 3.3: Visualization of Incremental Global Inference unrolled over time (from top to bottom). Depicted are two global decoding passes, and three trigger assignments in each pass (for two "jihad" and one "war" assignment). If two trigger assignments are on the same height, they are in the same sentence. Arrows represent global features. They point in the direction of the last known assignments of a word. Null assignments (∅) are excluded from global features.

figurations for the document. The algorithm is divided in two parts, *collection* and *prediction*.

In *collection* (Lines 2-4), IGI iterates m times through the document and calls GLOBAL BEAM SEARCH, a version of BEAM SEARCH (Section 3.1.1) which takes r into account, by drawing global features from the top configurations it contains. r is updated incrementally: In each pass, old top configurations are replaced with new (possibly refined) versions. Note that in the first pass, $r = \emptyset$; in this case, the global and local beam search versions are identical. Note further that we do not update features during collection.

In *prediction* (Lines 5-6), IGI calls GLOBAL BEAM SEARCH and refines r a last time. If it was called in training mode, decoding weight updates where necessary. Incremental Global Inference employs early updates within the structured perceptron framework.

Figure 3.3 visualizes the process. Depicted are two Incremental Global Inference passes and three trigger assignments in two sentences. The first "jihad" occurs in one sentence, "war" the second "jihad" (which are depicted on the same level, but higher) in another. All three are ATTACK triggers. As outlined above, the word "jihad" occurs only twice in the training data as an ATTACK trigger. Both occurrences were labeled as non-events in the first pass (which is identical to the base system). In the

second pass, Incremental Global Inference draws global features from r, i.e., from all top configurations it collected during the last pass, depicted as arrows with labels. Some arrows point upwards to information from the previous pass, and some arrows point to information from the current pass. This depends on whether the respective configuration was already replaced with the top configuration from the current pass or not. "jihad" and "war" stand in an hypernymy/hyponymy relation (with jihad being a kind of war). Furthermore, the two "jihad" occurrences stand in a 'string match' relation. Both relations help IGI to correctly classify both "jihad" occurrences as ATTACK triggers.

3.2.2 New Features

In addition to base system features (Section 3.1.3), Incremental Global Inference uses new features developed specifically for global decoding. We again use the distinction between *static vs. dynamic* features (Section 2.4).

Table 3.1 summarizes the new features. The first column reports the feature type (static or dynamic), the second column reports the condition under which features are generated, and the last column reports the actual features. The features belong to two broad groups: The first captures semantic similarity between words and is used to encourage the assignment of the same label to the same word (one sense per discourse). The second captures semantic relatedness of a word to the semantic fields of all event types and is used to infer labels of words never encountered during training.

Base system features are concatenated with event types to adapt a feature template to each of these types. Our features are not instantiated for single event types because they capture type-independent information. For example, one of the features captures the semantic relation between two words – we hypothesize that it is more important to indicate relations like '*A* is hypernym of *B*' than '*A*, a MEET trigger, is hypernym of *B*, also a MEET trigger'. The first form abstracts from (possibly infrequent) event types.

3.2.2.1 New Static Features

The new static feature we introduce is based on so-called *hidden units* (HUs), a concept from SEMAFOR, a state-of-the-art frame-semantic parser (Das et al., 2014). HUs are very similar to Bronstein et al. (2015).

Feature type	Condition	Feature
Static	WORD(n) related to HU(TYPE(n))	(1) HU-relation exists (2) HU-relation
Dynamic	WORD(n)==WORD(m) and TYPE(n)==TYPE(m) and WORD(n)	strmatch_same_type
	WORD(n) related to $\{x \in$ HU(TYPE(n)) and x is trigger$\}$	HU-relation

Table 3.1: New features for a trigger assignment n. Dynamic features are for trigger assignment pairs (n, m), where assignment m may come from the entire document. The function WORD returns the word string of its argument, the function HU the hidden units of its argument, and the function TYPE the respective event type.

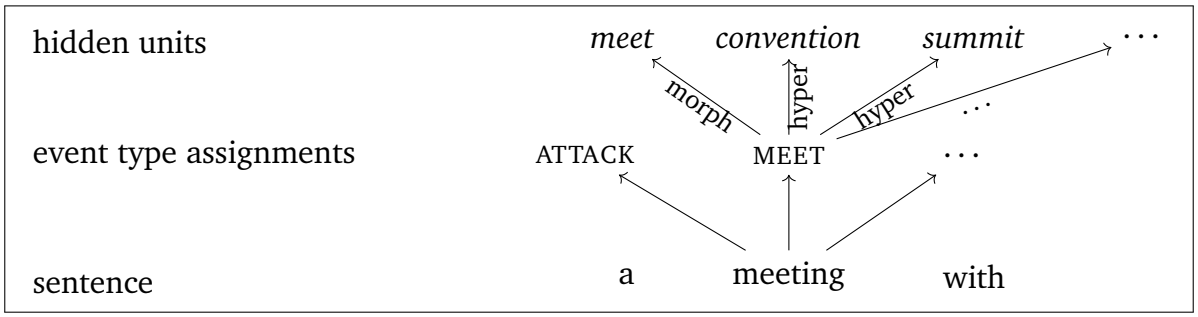


Figure 3.4: Visualization of our static hidden unit features. The first level of the figure is a part of a sentence. The word under consideration is "meeting". The middle level represent event type assignments. The upper part represents hidden units. Here, we have hidden units for "meeting" as a MEET trigger.

Identity, synonym, antonym, hypernym, hyponym, derivedForm, morphSet, verbGroup, entailment, entailedBy, seeAlso, causalRelation, sameNumber

Figure 3.5: WordNet relations used by our new local and global HU features.

The hidden units of an event type are the triggers we have seen in the training data for this type. HU features capture semantic relations of the word under consideration with all hidden units of an event type. For example, if the candidate trigger is *meeting* and the event type is MEET, we have several hidden units which share semantic relations with it: *convention* and *summit* as hypernyms, and *meet* as a morphological variant.

Figure 3.4 visualizes this. Here, we have the short phrase "a meeting with" and some exemplary event type assignments for "meeting". We know that 'meet', 'convention', 'summit' and others are hidden units of MEET triggers, and we also know the semantic relations holding between "meeting" and 'meet' (depicted 'morph'), "meeting" and 'convention' ('hyper'), "meeting" and 'summit' ('hyper'), and so on because we collected this information from the training data. For "meeting" as an ATTACK trigger for example we have no hidden unit features because "meeting" has no semantic relations with any hidden unit of ATTACK. If hidden units fire, they help IGI to determine some kind of similarity of a trigger candidate to the semantic field spanned of an event type.

We have a HU feature which indicates that there is a semantic relation between the word under consideration and at least one HU, and a feature which also includes the actual relation.

We use WordNet (Fellbaum, 1998) as our semantic resource. We do not consider all possible WordNet relations, but restrict them to the ones used by Das et al. (2014). Figure 3.5 reports all relations. The set includes actual semantic relations (hyponymy, antonymy, etc.) as well as syntactic relations (derivedForm, morphSet, etc.).

3.2.2.2 New Dynamic Features

We now describe the dynamic features for Incremental Global Inference. They are defined for a tuple of word-trigger type assignments (n, m), where n is the assignment under consideration and m may come from the entire document.

The first feature is based on the one-sense-per-discourse assumption. If a word triggers a specific event type, the same word tends to trigger the same event type within one document. Liao and Grishman (2010) found that this is true for 99.4%

of the ACE triggers. The respective feature fires if two trigger assignments share the same word and the same event type. This feature promotes trigger assignments which are in accordance to the one-sense-per-discourse assumption.

The second feature captures semantic relatedness of a word to all the triggers in a document. This feature is similar to the local HU features because it captures semantic relatedness of a word. The dynamic version however looks at semantic relations to all triggers in a document, instead of looking at relations to all known triggers of an event type. This feature has the ability to capture semantic relations to words which never occurred in the training data, and which are therefore not in any hidden unit set.

3.2.2.3 Missing Global Argument Features

The features described so far capture document-wide event-event interactions, i.e., global trigger relations. We were not able to devise features for global event argument relations. IGI improves argument identification and classification nevertheless because trigger performance, especially trigger recall, is considerably improved. We investigate argument classification performance of IGI's base system in Chapter 4. Here, we describe how IGI behaves with global argument inference. The following table compares IGI (Line 1) against a version with a global argument feature (Line 2) which fires if two coreferent entity mentions play the same role in matching event types.

	Trigger Classification			Argument Classification		
	P	R	F1	P	R	F1
IGI	71.0	70.7	70.8	64.5	50.9	56.9
IGI+arg	71.2	67.5	69.3	61.7	48.1	54.1

IGI with global argument features (Line 2) performs worse in terms of trigger and argument classification performance compared to not using global argument features (Line 1). The drop is 1.5 F_1 points for triggers and 2.8 F_1 points for arguments. All evaluation measures dropped in fact. This is a surprising results; we expected to see at least an increase in argument classification recall with global argument features.

The starting point of our global argument feature design is similar to the starting point of our global trigger feature design: Within one document, one entity plays one role for one event – which manifests itself on the textual level as: Coreferent entity mentions play the same role in coreferent events. Our global argument feature

however only incorporates entity mention coreference. Our experiments show that also incorporating event coreference (which is in itself a complex task) is crucial here.

The feature we test here – 'coreferent entity mentions play the same role in coreferent events – is effective because it captures an important characteristic of the event extraction task, at least in theory. Consider the following example.

(22) Officials say the pilot reported ice on the plane and planned to land in Massachusetts when the Plane left radar. ... The family had been heading to New Hampshire from Lakeland, Florida when their Plane went down.

The entity mention in question is "plane". It plays the same role in both TRANSPORT events (triggered by "land" and "heading"). In this case, the same entity plays the same role in events with the same type. The problem is that this feature starts to introduce wrong trigger assignments if it surpasses a certain weight. Consider the next example.

(23) The 7th Cavalry has pushed onward in the general direction of Baghdad. ... the 7th Cavalry came upon three Soviet vintage 20-millimeter anti-aircraft guns

Here, "7th Cavalry" is the entity mention under consideration. Suppose our global argument feature has a high weight. The system now favors that every possible trigger for "7th Cavalry" has the same type as the others, meaning that the system starts to wrongly favor a TRANSPORT label for "came" in order to enable the global argument feature to fire. Training tries to compensate by increasing the bias towards the non-trigger class ($\emptyset$), effectively dropping trigger recall, and this immediately lowers argument performance (because every missed trigger leads to missed arguments, see also Section 5.7.3), which training again tries to compensate by increasing the bias towards predicting arguments, which increases the amount of spurious predictions. A negative process which deteriorates overall prediction performance.

We can try to remedy this behavior during training by restricting the feature to coreferent events – 'two entity mentions play the same role in coreferent events'. The effect of this is even more severe because now we use gold information (about event coreference) which is not available during testing. The global argument feature with gold event coreference information receives a very high weight, because it only fires for correct event assignments during training. During testing, it strongly favors hypotheses

where the feature can be active, meaning that it again massively introduces spurious triggers. This is a good example why it is usually important to expose a system to its own errors during training.

Our problem with devising effective a global argument feature is that we cannot judge event coreference, we can only approximate it with the assumption that two events are coreferent if they express the same event type, which is clearly wrong. It seems beneficial to model event extraction and event coreference jointly.

3.2.3 Other Global Decoding Methods

In this section, we want to investigate IGI's relations to other global inference methods used in NLP literature. We focus on two methods in particular: the *Iterative Classification Algorithm* (Lu and Getoor, 2003; Sen et al., 2008; also strongly related to Neville and Jensen, 2000) and *Markov Logic Networks* (Richardson and Domingos, 2006; Poon and Vanderwende, 2010). Both methods exploit a graph structure to improve predictions for individual nodes. IGI falls into the same method class because it tries to improve the prediction of individual triggers based on their connection to semantically similar or related words.

The Iterative Collective Algorithm (ICA) leverages the structure of a graph to iteratively improve the classification of its nodes. It consists of two stages: bootstrap and iteration. Bootstrap assigns an initial labeling to all nodes given node features. Iteration updates node labels given their features *and* their incoming and outgoing edges until some termination criterion is met. For each stage a classifier is trained.

In principle, ICA can be applied to event extraction if we regard its predictions as some kind of (disconnected) graph. More precisely, we would have a bipartite graph with words and entity mentions as nodes, and syntactic and semantic relations (e.g., syntactic dependencies, WordNet relations, coreference links) as edges. Such a graph could be built for sentences or entire documents. ICA could first assign trigger and argument labels to all words and entity mentions, respectively, and refine them given the graph structure.

Incremental Global Inference and ICA are both *collective classification* algorithms. ICA's two-stage approach resembles Incremental Global Inference, where we also produce an initial classification of triggers and arguments and refine it given global features (Algorithm 4). However, there are important differences between the two.

First, to ensure that information from neighbor nodes are exploited optimally during training, ICA always relies on their gold labels. This means that, during training, the prediction of a node label is always based on the true labels of all other nodes, regardless of their predicted values. While this training regime makes optimal use of the graph structure during training, the model is faced with a very different situation during testing: predicted and noisy neighbor labels. However, since it never had to rely on wrong graph structure information, an ICA model tends to rely disproportionally on it. If the node labeling performance is not high enough, overall performance quickly degenerates for all predictions. IGI circumvents this problem by using predicted labels throughout, for training and testing. IGI can either be very cautious or heavily rely on related nodes, depending on the relation type. Exact string matches to hidden units for example (Section 3.2.2) have a big weight and tend to force the labels of the respective words to be the same; lemma matches have a rather small weight and influence the labels of related words less compared to exact string matches.

Second, ICA typically uses only simple 'edge features' like link counts (Lu and Getoor, 2003; Sen et al., 2008). In contrast, IGI uses more complex features like the semantic relation type.

An apparent difference is that ICA uses two, while IGI uses only one classifier. Lu and Getoor (2003) report that training two logistic regression models (one for node attributes, one for graph structure) and combining their predictions via a joint *maximum a posteriori* (MAP) estimate gives significantly better results on three document classification datasets compared to using one logistic regression model which takes node attributes and graph edges into account. However, IGI in its current formulation and implementation is based on the perceptron, a non-probabilistic, non-normalized learner. Combining node and graph perceptrons via a MAP equivalent is the same as having one perceptron with both feature sets. In every case, two perceptron scores are summed. This means that for IGI (and for perceptrons in general), it is equivalent to train two perceptrons and combine their scores or to train one with both feature sets.

Collective classification algorithms like ICA or IGI aim to refine an initial classification based on collective, global information. Markov Logic Networks (MLNs) on the other hand model a globally optimal labeling directly. Riedel et al. (2009) use MLNs for the BioNLP'09 event extraction shared task. Poon and Vanderwende (2010) further increase performance of MLNs on the same dataset.

Markov Logic is a framework for global inference. The problem is stated in the form of tuples (F_i, w_i) and constants, where F_i are first-order logic statements and w_i

are weights. Constants are equivalent to feature values. Logic statements are either defined over constants by forming atomic or compound feature templates, or they impose restrictions on the inference problem. In event extraction for example constants could be trigger and argument candidates as well as their lexical, semantic, or syntactic properties, and (instantiated) statements could either encode simple or complex features like 'if word==attack then label=ATTACK', or they could encode constraints like 'ATTACK has no Adjudicator role'. Each such instantiated statement has an associated learnable weight[5]. Inference assigns probabilities to 'possible worlds'. A possible world is a consistent instantiation of all logic statements (features and constraints) and their constants, e.g., a possible labeling of all trigger and argument candidates within a sentence or within a document. This framework is powerful in its expressiveness. However, it quickly becomes intractable to learn with increasing number of constants. Very efficient (and often proprietary) solvers have to be used to keep learning tractable.

3.3 Experiments and Results

In the following we describe the evaluation we devised for Incremental Global Inference. Please refer to Section 2.2.6 for a description of the train-dev-test split and to Section 2.4 for criteria when trigger and argument decisions are correct. Specificaly, in this section we evaluate

- **Experiment 1**: micro-averaged ACE performance using gold entities (standard setting)
  - **Experiment 1a**: IGI vs. the most recent reported event extractors
  - **Experiment 1b**: IGI vs. other systems with global inference
- **Experiment 2**: micro-averaged ACE performance using predicted entity mentions
- **Experiment 3**: micro-averaged TAC performance

[5]This means that constraints can be soft.

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Li et al. (2013)	76.9	65.0	70.4	73.7	62.3	67.5	69.8	47.9	56.8	**64.7**	44.4	52.7
Chen et al. (2015)	**80.4**	67.7	73.5	75.6	63.6	69.1	68.8	51.9	59.1	62.2	46.9	53.5
Nguyen et al. (2016)	68.5	75.7	71.9	66.0	**73.0**	69.3	61.4	**64.2**	62.8	54.2	**56.7**	55.4
IGI	73.7	**73.4**	73.6	71.0	70.7	**70.8**†	69.4	54.8	61.3	64.5	50.9	**56.9**†
Zhang et al. (2017)	n/a	n/a	n/a	75.1	64.3	69.3	n/a	n/a	n/a	63.3	50.1	55.9
Chen et al. (2017)	79.7	69.6	**74.3**	**75.7**	66.0	70.5	**71.4**	56.9	**63.3**	62.8	50.1	55.7

Table 3.2: Micro-averaged precision, recall, and F_1 for Experiment 1a: Incremental Global Inference (IGI) vs. five of the most recent event extractors. Numbers in bold are the best for the respective measure. Evaluation numbers published after IGI are reported in the lower half of the table. † means statistically significant compared to Li et al. (2013) at the $p < 0.05$ level. We only measured significance for 'classification' F_1 scores.

3.3.1 Experiment 1: ACE 2005, Standard Setting

Experiment 1 compares Incremental Global Inference against other systems in what we call the 'standard setting' of ACE event extraction: using gold entity mentions (Li et al., 2013; Chen et al., 2015; Nguyen et al., 2016, i.a.) and evaluating on the data split introduced in Ji and Grishman (2008). Experiment 1a evaluates the method against the best reported systems. Experiment 1b evaluates it against other global inference methods for event extraction.

Table 3.2 reports evaluation results for Incremental Global Inference (Line 4) and five other systems (Experiment 1a), two of which were published after IGI (Lines 5-6). Line 1 (Li et al., 2013) is our base system.

In terms of F_1, IGI performs better than all systems, including recent deep learning approaches (Chen et al., 2015; Nguyen et al., 2016; Zhang et al., 2017; Chen et al., 2017). It provides the best balance between precision and recall for trigger classification, resulting in the highest trigger classification F_1 for a full event extractor.

Compared to the base system, IGI has a better F_1 score in all evaluation categories. It improves trigger identification by 3.2, trigger classification by 3.3, argument identification by 4.5, and argument classification by 4.2 F_1 points. We measured significance for the 'classification' F_1 scores – the differences are statistically significant at the $p < 0.05$ level.[6] This increase is due to a much higher trigger recall (+8.4 points) compared to the base system, without losing too much precision. Nguyen et al. (2016)

[6]We measured significance using approximate randomization (Noreen, 1989).

	Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1
Ji and Grishman (2008) cross-sent	64.3	59.4	61.8	54.6	38.5	45.1	49.2	34.7	40.7
Ji and Grishman (2008) cross-doc	60.2	**76.4**	67.3	55.7	39.5	46.2	51.3	36.4	42.6
Liao and Grishman (2010)	68.7	68.9	68.9	50.9	49.7	50.3	45.1	44.1	44,6
Hong et al. (2011)	**72.9**	64.3	68.3	53.4	52.9	53.2	51.6	45.5	48.4
IGI	71.0	70.7	**70.8**	**69.4**	**54.8**	**61.3**	**64.5**	**50.9**	**56.9**

Table 3.3: Micro-averaged precision, recall, and F_1 for Experiment 1b: Incremental Global Inference (IGI) vs. four systems which are not limited to intra-sentential inference. Numbers in bold are the best for the respective measure. Yang and Mitchell (2016) is not reported here because they use predicted entity mentions.

is the only system with a higher trigger classification F_1 than IGI. However, it also has a considerably lower precision.

The increased argument performance of IGI is enabled by the increased trigger recall it provides – a system which increases recall without decreasing precision too much has a good chance to also increase argument performance, because it has a chance to correctly predict arguments for each additional trigger it finds. This can be seen for IGI, but also for Nguyen et al. (2016). We test this hypothesis in Section 5.7.3 where we find that there is a moderate positive correlation between trigger recall and argument F_1.

Compared to the best system published before IGI (Nguyen et al. (2016), Line 3), our approach improves trigger identification by 1.7, trigger classification by 1.5, and argument classification by 1.5 F_1 points.[7]

Table 3.3 reports evaluation result for IGI (Line 5) and four other systems which use some kind of inter-sentence decoding. Ji and Grishman (2008) use either cross-sentence (Line 1) or cross-document decoding (Line 2). Liao and Grishman (2010) (Line 3) is a system which uses multiple classifiers to perform document-wide inference. Finally, Hong et al. (2011) use cross-entity inference to predict event arguments.

In this evaluation setting we cannot report *trigger identification* performance because it was not reported in the other publications.

[7]Nguyen et al. (2016) report the highest argument recall, with a wide margin to the second-highest value. We speculate that the improvement here is either a result of their word embeddings, or the result of a recall-driven optimization procedure. However, the authors give no insight into what caused their high recall.

We can see that IGI outperforms all previous systems with global inference capabilities by a wide margin. It improves trigger classification by 2.5, argument identification by 8.1, and argument classification by 8.5 F_1 points.

From a scientific point of view, it is difficult to compare different global decoding procedures themselves because their performance heavily depends on the performance of their base systems.

Liao and Grishman (2010) and Hong et al. (2011) have a higher trigger classification F_1 than our base system (+1.4 and +0.8, respectively). However, IGI trigger classification F_1 outperforms both (+1.9 and + 2.5, respectively), indicating that our method is superior.

The high argument performance improvement is partially due to the better argument extraction performance of our base system (4.3 F_1 points better than Hong et al.), and partially due to the increased trigger performance.

3.3.2 Experiment 2: ACE 2005, Predicted Entity Mentions

We now present evaluation numbers for IGI using predicted entities. These numbers are published in Judea and Strube (2016).

The main difference between Experiments 1 and 2 is the handling of entity mentions. While Experiment 1 (standard setting) allows to use gold entity mentions, Experiment 2 requires to predict them. Entity mention prediction constitutes a difficult problem in itself. ACE defines seven entity types which can be used as event argument fillers. Refer to Section 2.2.4 for a detailed discussion of ACE entity mentions.

To evaluate Incremental Global Inference in this setting, we use a reimplementation of Li et al. (2014) as our base system. In contrast to the base system we used so far, a base system for Experiment 2 not only enumerates possible triggers at each token position, but also possible entity mentions. The new base system generates hypotheses which not only encompass triggers, but also entity mentions of varying length. On top of the hypotheses building described in Section 3.1, this base system also performs the following actions. At each token position i which is not a trigger in the current hypothesis, this base system enumerates segments (entity mention candidates) of increasing length ending at this position. Each segment is paired with each entity type, each such pairing constituting a new hypothesis. For a segment-type pair, we generate the features reported in Li et al. (2014). A maximum segment length for each entity type is collected from the training data. Li et al. (2014) omit the generation of values

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
Baseline	**67.2**	62.2	64.6	**64.7**	59.9	62.2	70.7	36.6	48.2	57.3	29.7	39.1
IGI	67.0	**65.9**	**66.5**	64.2	**63.1**	**63.7**	**76.1**	**40.8**	**53.1**	**59.9**	**32.1**	**41.8**

Table 3.4: Micro-averaged precision, recall, and F_1 for Experiment 2: Incremental Global Inference (IGI) vs. our base system, both using predicted entity mentions. The base system is a reimplementation of Li et al. (2014). Numbers in bold are the best for the respective measure.

(e.g., `Crime`) and times because "they can be most effectively extracted by handcrafted patterns." We follow them in this regard.

Table 3.4 reports micro-averaged precision, recall, and F_1 for Experiment 2. We compare a baseline (Li et al., 2014) to Incremental Global Inference using predicted entity mentions. Please note that we evaluate on the same test set as Li et al. (2014) to ensure comparability; this test set differs from the more widespread test set used in Experiment 1.

Here, we see the same trends as in Experiment 1: IGI leads to increased trigger prediction recall, which also has a positive effect on argument prediction. The relative increase in trigger classification F_1 is on the same scale as in Experiment 1: About 1.5 points improvement. Argument classification F_1 increased a bit more in Experiment 2, namely 2.7 F_1 points.

3.3.3 Experiment 3: TAC 2015, Standard Setting

Experiment 3 evaluates Incremental Global Inference on different annotations than Experiments 1 and 2, namely on TAC data. The TAC 2015 data set was released in the context of the TAC shared task. It uses a different, but related annotation scheme than ACE 2005. Please refer to Section 2.2 for a detailed description of the TAC 2015 dataset and its relations to ACE 2005. The numbers presented here were published in Judea and Strube (2016).

The biggest differences between ACE and TAC (apart from different event types) is that TAC includes multiple event labels per word. For example, the word "murder" usually triggers both, an ATTACK and a DIE event. Our system cannot deal with such cases. For training, we randomly drop one of the labels. For evaluation, we include all labels, meaning that we always produce one error in multi-label cases.

	Trigger Identification			Trigger Classification		
	P	R	F_1	P	R	F_1
Baseline	**87.3**	47.0	61.1	**73.3**	39.5	51.3
IGI	77.1	**54.7**	**64.0**	64.1	**45.5**	**53.3**

Table 3.5: Micro-averaged precision, recall, and F_1 for Experiment 3: Incremental Global Inference (IGI) vs. our base system on the TAC 2015 dataset. Numbers in bold are the best for the respective measure.

Table 3.5 reports precision, recall, and F_1 for trigger identification and classification on the TAC 2015 data set (Experiment 3). Again, IGI increases trigger recall and consequently trigger F_1 considerably compared to the local baseline.

3.4 Error Analysis

We begin error analysis by a confusion heat map depicted in Figure 3.6. An analogous heat map for the base system is depicted in A.1. The heat map uses a darker color for higher values. The labels on the y axis are the true labels (or gold labels), and ones on the x axis are the predicted labels. Cells contain frequencies. We also report the sum of rows and columns in parentheses after the event type in addition to frequencies. For example, there are 6 END-ORG events in the test set, 5 of which were classified correctly and one which was confused with START-ORG. The label "null" represents $\emptyset$, or 'no event'.

From the diagonal of the heat map (which represents correct assignments) we can see that ATTACK, MEET, and TRANSPORT events have the highest amount of correct classifications. They belong to the most frequent event types in ACE (Section 2.2.6). However, DIE, which is in the overall data the third most frequent event type, is underrepresented in the test set, and MEET is a bit overrepresented.

IGI rarely confuses two event types, the vast majority of errors are confusions with $\emptyset$. IGI has nearly as many false positives (123) as false negatives (133). A false positive is given when the system predicted an event type, but the true label is $\emptyset$. A false negative is the exact opposite case. We will now discuss errors specific to global inference.

In some cases like "fighting running battles", IGI assigns the ATTACK label to "battles" (as it does for "fighting") – not a wrong decision per se. However, a wrong decision ac-

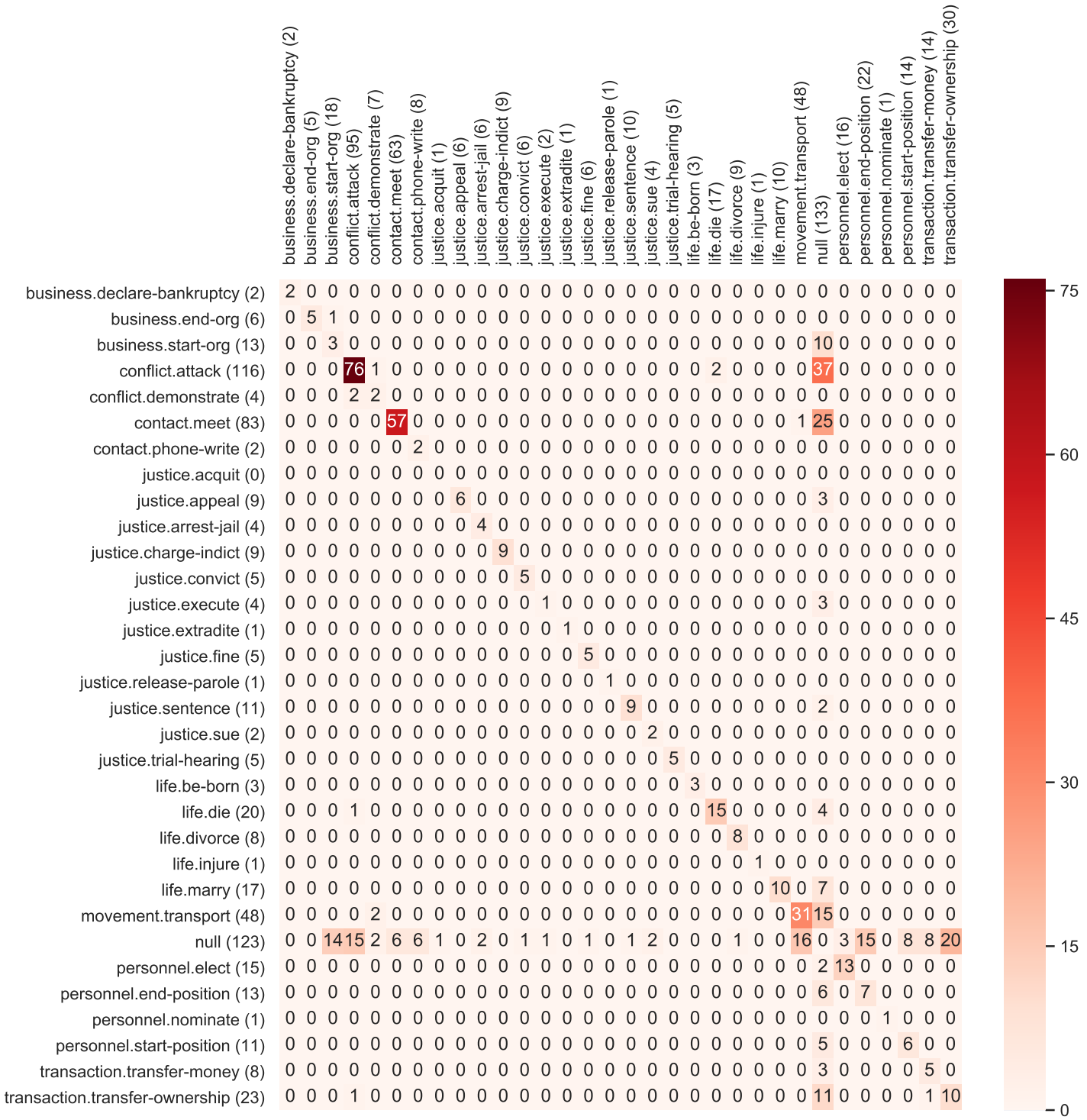


Figure 3.6: A heat map (darker colors mean higher values) representation of the confusion matrix for Incremental Global Inference. Gold labels are on the Y axis, predicted on the X axis. In Appendix Figure A.1, we depict an analogous heat map for the base system.

cording to the ACE annotation guidelines which forbid annotating two triggers within one sentence for the same event.

Sometimes, an alleged trigger is part of a name, e.g., "Win Without War", where the final word was confused with an ATTACK trigger. Many such cases can be excluded as error sources in the standard evaluation setting with gold entity mentions because they tend to be annotated as organizations and are therefore excluded from the set of possible event triggers.

Another error source is polysemy. Compare the sentences "fire on the Palestine hotel" vs. "killed by friendly fire" from one document. The second "fire" was mistakenly classified as an ATTACK trigger because the first was classified as such within the same document. This is a case where the one-sense-per-discourse assumption leads to a wrong classification. However, this can (arguably) also be regarded as a case where the annotators overlooked an annotation.

Global features can sometimes be clearly misleading. Consider the following sentences: "Toyoda, who joined Toyota" and "He joined the board". Here, only the first sentence indicates a START-POSITION event, the second however does not trigger any ACE event. IGI tags also the second case as a START-POSITION trigger because the features which fire here (most notably exact string match) are strong indicators that the same event type should be assigned to both. The only way to distinguish such cases is to look at syntactic relations. In one case, "Toyota" is the object of the potential trigger, in the other, it is "board". Only in the first case the object is an entity mention with an appropriate entity type, *org*. Theoretically, IGI's joint inference can handle such cases, because they are covered by appropriate features (e.g., 'which entity type does the object of the potential trigger have'). However, global features often outvote the respective local features because of their higher weights. In our example, the exact string match feature has an especially high weight. Future research has to investigate ways to incorporate more context sensitivity into global features.

Figurative speech may be challenging for IGI as well. For example, in "Blair faces an uphill battle to win Bush over", "battle" is used as a metaphor and does not indicate an ATTACK event as the system predicted. Such cases are not only difficult to IGI, but to any event extractor, because they look very similar to actual ACE event triggers, and even their contexts do sometimes not bear enough information to revise the decision for a strong ATTACK trigger like "battle".

3.5 Conclusion

Trigger prediction is highly sensitive to lexical features (either categorical or in form of word embeddings). The fact that our method outperforms deep learning methods without word embeddings suggests that a global, document-wide inference is complementary and might also benefit these systems. We outline and continue this thought in Section 7.2, where we sketch a system which uses IGI in combination with a neural event extractor based on syntax encoders (Chapter 5).

IGI can work with different base systems to produce the initial labeling – we use it with the base system (Section 3.1.1) as well as with a variant which uses predicted entity mentions (3.3.2); in both cases, IGI adopts the capabilities of both base systems. Similarly, it might be used in more recent 'neural' event extractors as base systems, e.g., with our neural syntax encoders (Chapter 5). We discuss this point in Section 7.2 in more details.

Systems with a high trigger classification recall have a good chance to show increased argument classification performance merely because of the additional triggers they find, and not because they actually predict arguments better. We show in Section 5.7.3 that there is a positive correlation between trigger recall and argument performance across multiple methods and data splits. IGI considerably improves trigger classification recall (and performance in general), and this in turn leads to a considerably increased argument classification performance because IGI is able to actually perform argument classification for more correct triggers compared to the base system. This might also hold for Nguyen et al. (2016). They have a very high trigger classification recall paired with a low precision – however, they report a good argument classification F_1. Zhang et al. (2017) and Chen et al. (2017) on the other hand have lower trigger classification recalls, but an argument performance comparable to Nguyen et al. (2016). This may be an indicator that the latter two can predict arguments better than the first. However, it is hard to disentangle trigger and argument performances because of their strong interdependencies.

We could not devise effective global features which help argument prediction directly (Section 3.2.2.3). In Chapter 4, we look closely on argument prediction, especially on difficulties and possible solutions. We devise experiments to evaluate argument performance in isolation, without any interference from trigger prediction.

4 Syntax Encoding for Event Argument Classification

In this chapter, we analyze event argument identification and classification – the task of identifying entity mentions which are arguments of a given event and assigning the role they play in the event (Section 2.4). Based on our findings, we explore distributed syntax representations. We build a system which learns to represent syntactic structures for event argument identification and classification.[1] The ultimate goal of this chapter is to investigate how to improve argument classification performance on the one hand, and how to make better use of syntax information on the other. We use the knowledge acquired here to build a syntax-based event extractor in Chapter 5.

When we look at the evaluations in three of the most influential event extraction papers (Li et al., 2013, 2014; Chen et al., 2015; Nguyen et al., 2016) we note that argument classification performance is low, ranging from 52.7 to 55.4 F_1. There are multiple reasons for this.

First, argument classification suffers from error propagation. Missed or spurious event triggers may lead to missed or spurious event arguments. Second, event structure is complex. Multiple entities can play the same role in the same event. Additionally, one entity can play different roles across events (and thus cause multiple event arguments). Consider the following example.

> A Palestinian `Boy` as well as his `Brother` and a `Sister` were wounded late Wednesday by Israeli gunfire.

Here, the three entity mentions "boy", "brother", and "sister" are all `Victims` of the INJURE event triggered by "wounded", as well as `Targets` of the ATTACK event triggered by "gunfire". Such structures can become even more complex when more events and more entities are involved. See Chapter 2 for a more thorough discussion.

[1] In the following, we will only write about argument classification, but mean both tasks.

The third reason for low argument classification performance is syntactic complexity. Many arguments are syntactically far away from their triggers, making it hard to construct meaningful syntactic features. Section 4.1 shows that performance is tied to the length of the shortest dependency path connecting trigger and argument.

The first two error sources are addressed by joint inference: If we assign entire event structures to sentences, we have no error propagation because trigger and argument decisions influence each other. Additionally, it becomes easier to leverage the fact that the roles one entity plays in related events are coherent: The `Victim` of one event for example can also be a `Target` in another, but it is probably never the `Attacker`. In joint inference over entire event structures, we can have features which capture the roles an entity mention plays in all the events within a sentence.[2] This line of work is investigated in more detail in Chapter 3.

To the best of our knowledge, no previous work identified syntactic complexity as the third key problem for argument classification performance, and no previous system decomposes syntactic structure in order to learn better classifiers for the task. Our contributions in this chapter are the following.

1. We observe that syntactic complexity is a crucial factor for argument classification – performance highly correlates with dependency path length (Section 4.1).

2. We represent dependency paths (Section 4.2) with Recurrent Neural Networks (Section 4.3.3) in order to account for their sequential and compositional nature. Using RNNs to learn dependency path representations proved effective in other areas like relation extraction (Xu et al., 2015b) and semantic role labeling (Roth and Lapata, 2016). We investigate their use for argument classification.

3. We represent lexical contexts of event arguments with Convolutional Neural Networks (Section 4.3.4). Together with RNNs, they form an effective and simple argument identifier which beats a state-of-the-art, feature-based system without any manual feature engineering.

Section 4.1 analyses event argument classification performance for our baseline. Section 4.2 presents lexicalized shortest dependency paths and consider the most important difficulties. Section 4.3 presents the problem formulation and the system ar-

[2] Note that such features have to use coreference resolution (Martschat and Strube, 2015) to determine which mentions belong to the same entity.

chitecture. Section 4.4 reports our experiments and the respective results. Finally, Section 4.5 gives the conclusion.

4.1 Baseline Performance Analysis

Here, we analyze the performance of our base event extractor form Chapter 3 (Li et al., 2013, 2014) with regard to argument performance. We decided to use this system because it is still one of a few joint event extractors, making it a challenging baseline and an apt candidate for a good performance analysis. In the following, we omit a description of this system. Please refer to Section 3.1 for a thorough presentation. However, we briefly mention its argument features below.

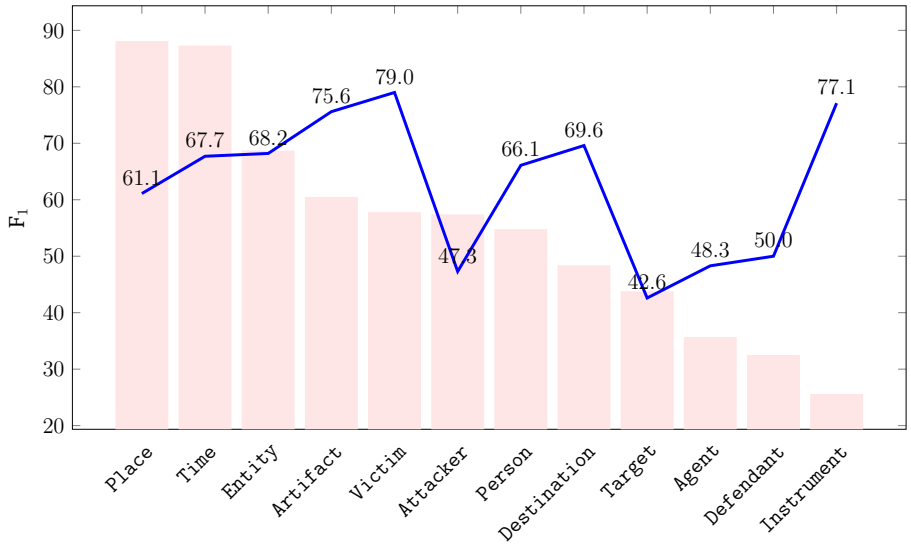
The baseline system uses a rich, hand-engineered feature set to predict event arguments. The set can be divided into features produced by static and dynamic feature templates.

Static feature templates characterize a single argument, and they involve only the entity mention and the trigger of this argument. They capture, e.g., the trigger and entity types, the words of the potential argument, its lexical context, and the dependency path connecting it to the trigger.

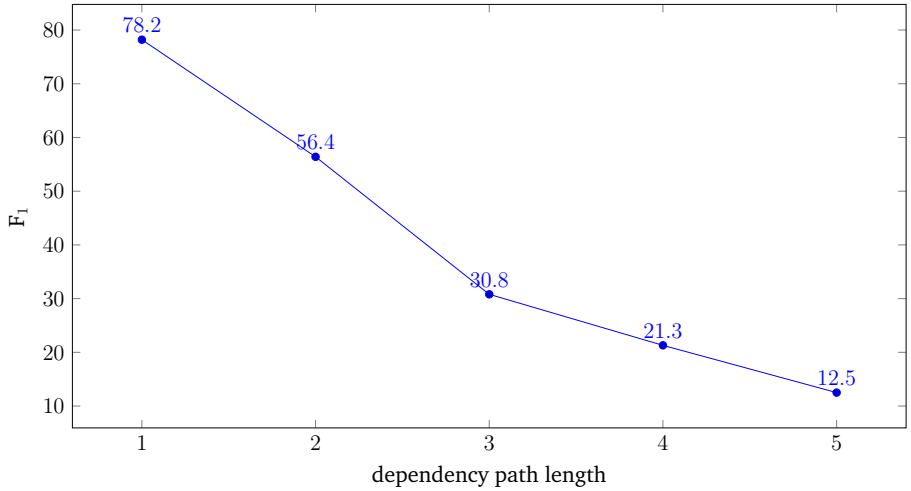
Features produced by dynamic templates characterize multiple arguments in terms of the entity mentions and roles they share. They also capture characteristics of the entity mentions within one event, e.g., the words between two entity mentions sharing a role in one event, or the head and modifier of nominal modifications like 'IBM' in 'IBM CEO'. Other templates capture characteristics of events with shared entity mentions, e.g., all the roles it plays in events. The system uses a total of two dozen feature templates, resulting in 150,000 features for argument classification.

Below, we analyze its performance before we propose a new system which specifically targets one of its blind spots – long dependency paths which connect triggers and their arguments.

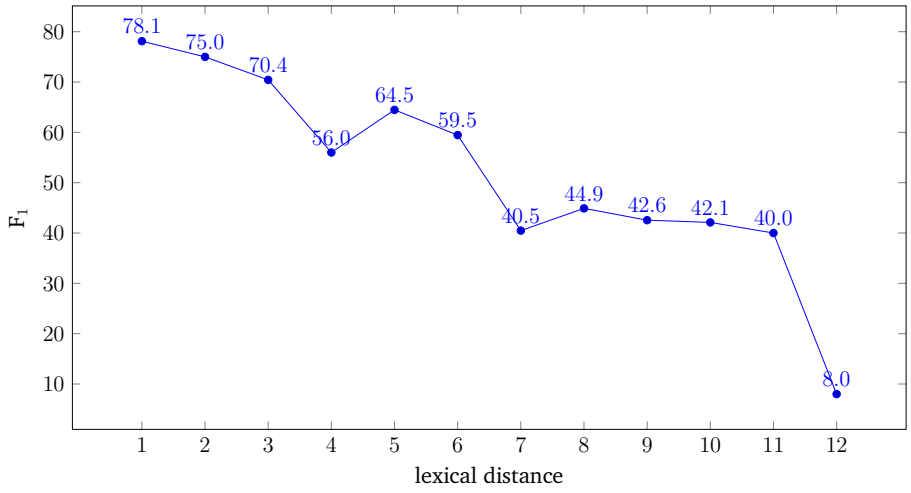
We start the analysis of argument classification performance with the observation that despite the low overall scores, some argument types perform reasonably well. Table 4.1 reports development set F_1 of our baseline for the three best and the three worst performing argument types in the development set.



(a) Decreasing trainset support of the 12 most frequent argument types plotted against baseline devset F_1. Bars represent scaled support.



(b) Increasing dependency path length plotted against baseline devset F_1.



(c) Increasing lexical distance plotted against baseline devset F_1.

Figure 4.1: Training set support, lexical distance of trigger and argument, and dependency path length plotted against development set performance.

Argument type	$\text{Support}_{\text{train}}$	$F_{1\,\text{dev}}$
`Victim`	578	79.0
`Instrument`	256	77.1
`Artifact`	605	75.6
`Agent`	357	48.3
`Attacker`	574	47.3
`Target`	438	42.6

Table 4.1: Training set support and development set baseline F_1 for the three best and three worst performing argument types. We excluded types with less than 20 samples in the development set.

Performance for the upper half of argument types is quite good (77 F_1 points on average), while performance for the lower half is worse (40 F_1 points on average). What is the reason for this big difference?

Our first assumption is that the best performing types have more training data. Indeed, the best-performing type `Victim` has considerably more samples than the worst-performing type `Giver`. `Attacker` however has nearly the same amount of training samples as `Victim` but a much lower performance (-31.7 F_1). `Instrument` on the other hand has only half the training samples but a better performance (+29.8 F_1).

To further investigate this, Figure 4.1a plots training set support in decreasing magnitude against development set F_1 for the 12 most frequent argument types. The plot is not conclusive: More training data does not automatically lead to better performance. The most frequent argument type `Place` with 881 training samples has an F_1 of 61.1, whereas `Victim` with 34% less training samples has an F_1 of 79.0. `Instrument` has about 70% less training samples and an F_1 of 77.1. If the number of training samples is not an important factor for performance, what else could be?

One answer is semantic variety: Some roles can only be filled by one or two entity types, and many of the respective entity mentions *are* role fillers. This is especially true for `Instrument` which can only be filled by *vehicles* and *weapons* in ACE 2005; in turn, many *weapons* are `Instruments`. This is reflected in the good performance of `Instrument` in Table 4.1 and Figure 4.1a.

However, most roles can be filled by more than two entity types, and their potential fillers are more frequent than *vehicles* and *weapons*: `Entity` for example can be filled by *persons*, *organizations*, and *geopolitical entities*. At the same time, most of the respective mentions are *not* `Entities`. Even if a role can only be filled by one entity

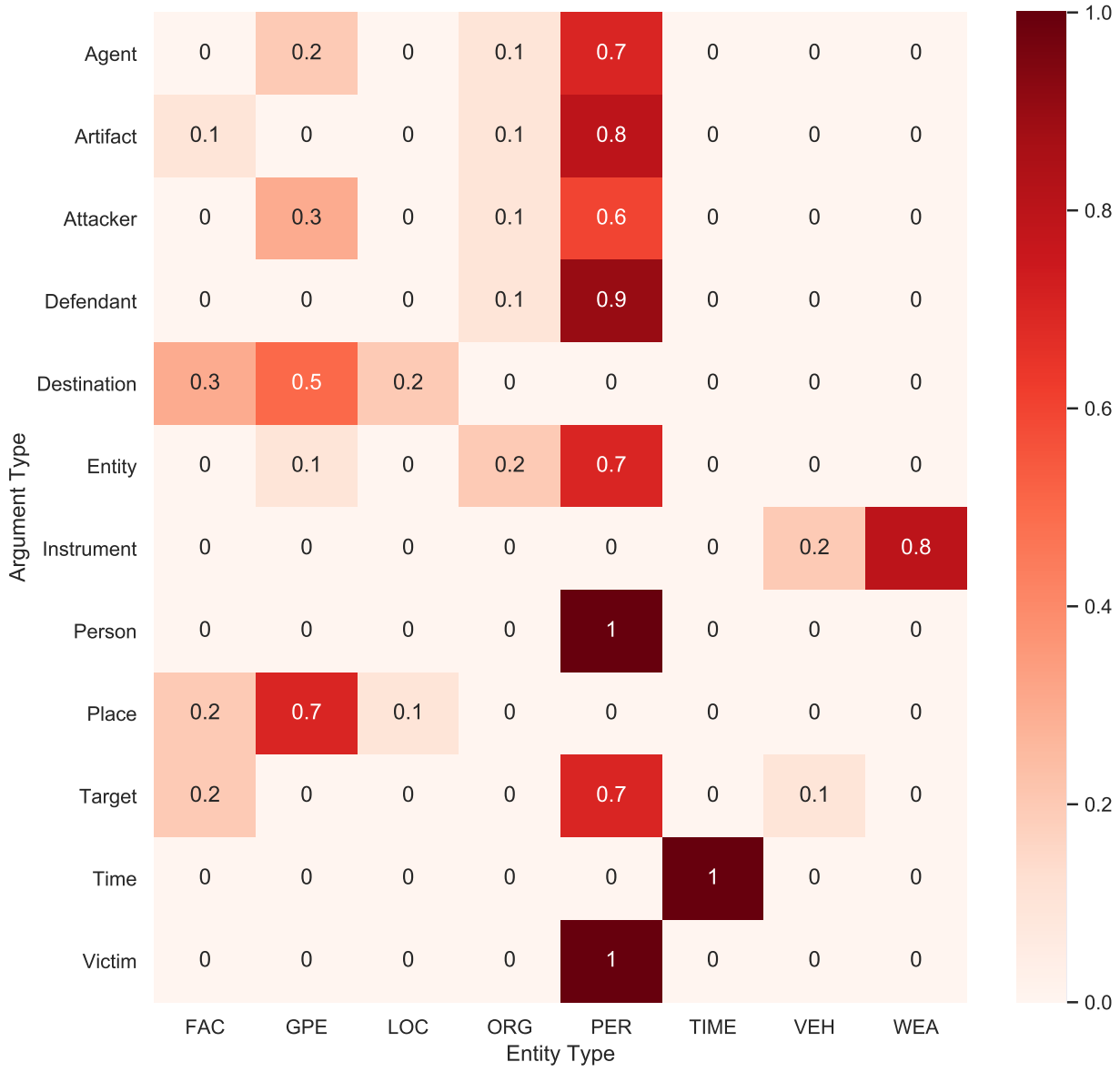


Figure 4.2: A heat map (darker colors mean higher values) of the entity type distribution for the twelve most frequent argument types. Values in rows sum up to 1.

Type	Count	Argument Count	Ratio
FAC	1797	453	0.25
GPE	9454	1301	0.14
LOC	1363	264	0.19
ORG	6875	421	0.06
PER	33278	3496	0.11
TIME	5469	1095	0.20
VEH	1048	146	0.14
WEA	1009	253	0.25

Table 4.2: Entity types with total occurrences ("Count"), as event arguments ("Rolecount"), and a ratio of occurrences as arguments to total occurrences.

type, it may be that most occurrences are not role fillers, making it harder to correctly fill those roles. Consider `Time` for example, which can only be filled by *time* mentions, yet it has only a mediocre performance of 67.7 F_1 points on the development set. Semantic variety alone cannot explain the big performance differences between argument types. To make this point more clear, Figure 4.2 depicts a heatmap of entity type distributions for the twelve most frequent argument types, and Table 4.2 reports how many entity mentions of the respective type occur, how often they are event arguments, and a ratio between the two (high values mean frequent occurrence as event arguments). The figure and the table report complementary information. For example, 80% of `Instrument` fillers are *wea* (weapons) and 20% are *veh* (vehicles). In fact, WEA can *only* fill `Instrument` roles. In turn, 25% of all *wea* mentions are `Instruments`.[3]

`Time` roles can also only be filled by one entity type, namely *time*, and 20% of all *time* occurrences are role fillers. Even though we have a similar semantic variety as with *wea* and `Instrument`, the performance of `Time` is about 10 F_1 points lower than that of `Instrument`. Semantic variety alone cannot explain the performance differences.

Another important factor is syntactic complexity: How long and diverse are dependency paths connecting arguments and triggers? To investigate the effect of syntactic complexity, Figure 4.1b depicts length of dependency paths connecting triggers and arguments in decreasing magnitude against development set F_1. In this plot, we see a much clearer trend: Shorter syntactic distance leads to better performance. Length-

[3] If we count the number of *wea* mentions in sentences with events which have an `Instrument` role, the percentage is most likely considerably higher.

1 paths (direct trigger-argument dependency) have an F_1 of 78.2. Length-2 path F_1 drops to 56.4, and to 30.8 for length-3 paths. Length-4 and length-5 paths have an F_1 of 21.3 and 12.5, respectively.

This trend is also reflected in the performance of individual argument types. The most frequent type `Place` has a high average path length of 2.2 and a low F_1 of 61.1. `Victim` on the other hand has considerably less training data, but an average path length of 1.5 and an F_1 of 79.0. And `Time`, the example we discussed above, has an average path length of 1.9. For the three best performing types, the average path length is 1.7 vs. 2.3 for the three worst performing types.

Dependency path length is related to lexical distance – the longer a dependency path, the more words are usually between trigger and argument. To investigate the effect of lexical distance, Figure 4.1c depicts the number of words between trigger and argument in increasing magnitude against development set performance. Here, we see a somewhat less clearer trend: Increasing lexical distance leads to lower performance, with a considerable increase between distances 4 and 5, and a performance plateau between 8 and 11.

We decided to favor syntactic over lexical structure because dependency paths abstract from the actual words and ignore many which are irrelevant for argument classification, like adjectives and adverbs. This in turn alleviates data sparsity. A dependency path like returned$\xrightarrow{nmod:from}$summit$\xrightarrow{nmod:in}$Ireland is more concise and relevant for the task than a word sequence like “returned from a summit in Ireland”.

We conclude that syntactic complexity is a crucial factor for argument classification. Therefore, it is inevitable to reduce or better handle syntactic complexity. Most systems incorporate dependency paths merely as strings, or rely on direct dependencies of triggers and arguments. They do not decompose or further analyze dependency paths in order to find relevant substructures, or to deal with data sparsity of long paths. In Section 4.3, we present a simple and efficient system which directly addresses syntactic complexity. It will mainly operate on dependency paths between potential arguments and triggers.

4.2 Dependency Paths

We will present dependency paths in more depth in this section. We first start with a definition of 'dependency path' and illustrate afterwards the benefits and difficulties of using dependency paths for argument classification.

Dependency paths are paths in a graph of grammatical dependencies. A dependency is a triple $(w_1, d^{1:2}, w_2)$ where w_1 is called the *governor*, w_2 the *dependent*, and d is a grammatical relation like nsubj (subject) or dobj (direct object), further specified by the word indices it connects. Grammatical dependencies are based on the work of De Marneffe and Manning (2008) and Schuster and Manning (2016) (enhanced++ dependencies). Each dependency triple connects two words in a sentence with a grammatical relation; however, governor and dependent may also participate in relations with other words. In total, a relation graph is formed. We will use this graph to compute grammatical paths which connect triggers with potential arguments.

Dependencies as formalized above are *directed*. Paths of directed triples are also directed. In such a directed graph however, there may be no path between two words. For this reason, we do not distinguish between governor and dependent position in dependency relations. We will encode the direction of a relation by introducing two dependency labels for each original label, denoted with a $\leftarrow$ and $\rightarrow$ suffix.

A dependency path consists of tuples as defined above. More formally, a dependency path $P_{w_1 \rightarrow w_n}$ between words w_1 and w_n is defined by

$$P_{w_1 \rightarrow w_n} = (w_1, d^{1:2}, w_2, \ldots, d^{n-1:n}, w_n). \tag{4.1}$$

Our dependency paths are lexicalized, meaning that we include the words which participate in a grammatical relation. We explicitly call our paths lexicalized to distinguish them from sequences of dependency labels. Paths as defined here are input to the Long Short-Term Memory Network we describe in Section 4.3.3.3.

Our paths always start with the trigger word and end with a mention word (w_1 and w_n, respectively in the equation above).[4] We say that a path has length 1 if trigger and argument are directly related, length 2 if the path includes one intermediate dependency, etc.

Often, short dependency paths directly reflect event argument structure:

killed $\xrightarrow{\text{nsubj}}$ father-in-law (`Agent`)

[4]For multiword expressions, the path connects trigger and entity mention head.

killed $\xrightarrow{\text{dobj}}$ him (Victim)

The trigger "killed" (a DIE event) has two dependencies, "father-in-law" being the subject and "him" being the object. Even without looking at more context we can say with confidence that the subject must be the Agent of the event and the object must be the Victim. Even longer paths may be quite clear:

returned $\xrightarrow{nmod:from}$ summit $\xrightarrow{nmod:in}$ Ireland (Origin)

Here, "returned" triggers a TRANSPORT event. The path conveys the information that some entity returns from a summit in Ireland, making "Ireland" the Origin of the event.

Unfortunately, not all short paths clearly indicate an event argument and its role. The biggest problem is polysemy. A path like killed $\xrightarrow{dobj}$ him in the sentence 'the argument killed him' does not indicate a Victim, because the verb is used figuratively. Polysemy is not a problem for the system presented in this chapter, because we use gold triggers here to investigate event argument classification in isolation, but it is a problem for the full event extractor presented in Chapter 5.

Of course, not all dependency paths are as easy to interpret. The following examples show the necessity to decompose dependency paths in order to catch similarities between them.

war $\xleftarrow{dobj}$ fighting $\xrightarrow{nsubj}$ forces ("coalition forces fighting the war") (Attacker)
war $\xleftarrow{nmod:to}$ go $\xrightarrow{nsubj}$ we (""we go to war") (Attacker)

To make the last two paths more readable, we included the phrases they encode. These paths are more complex than previous ones because trigger and argument are governed by other words, namely by "fighting" and "go". In both cases, "war" triggers an ATTACK event and the subject is an Attacker argument. Humans can easily spot similarities in the two paths. The arguments are in both cases the subjects of the governing verbs: "forces" is the entity fighting a war and "we" is the entity going to war. However, the left sides of the paths look quite different: In one case, "war" is the direct object of the governing verb, in the other it is a prepositional complement. A system must learn that "fighting the war" and "go to war" are roughly the same, even though "fighting" and "go" do not share meaning, and even though "war" is embedded by different syntactic constructs in the dependency tree. A system needs

the ability to decompose the paths and to learn the meaning of sequences of words and dependencies.

Another problem is given by the sole length of a path – long paths require long sentences, and parsing accuracy for long sentences is worse than for short sentences (Liu and Zhang, 2017).[5] Furthermore, long paths are more likely to contain elements never encountered during training. Even a short path like go$\xrightarrow{nmod:to}$war may be expressed as went$\xrightarrow{nmod:to}$ war or went$\xrightarrow{nmod:to}$fight, which are both not part of the training data.

In the following, we will present a system which uses at its core a sequence models to learn distributed representations of dependency paths. We will first state the problem and formalize it before we present and describe the system architecture. Section 4.3.3 describes how we represent dependency paths and how this information is used by the system to predict event arguments.

4.3 biLSTM/CNN: Problem Formulation and System Architecture

Input to our system (biLSTM/CNN) are instances as described in Section 4.3.1. An instance has three information groups, each is processed by one component. The first produces a representation of the event type, mention type, and text genre (Section 4.3.2). The second produces a representation of the lexicalized dependency path (Section 4.3.3). The third extracts valuable information from the lexical context (Section 4.3.4).

Figure 4.3 depicts the system architecture. The figure is split in four (bottom, middle left/right, and top), each part visualizing one of the components we describe and formalize below. We will first specify the input before we describe each component in detail. In the remained of this chapter, $\oplus$ means the concatenation of vectors.

4.3.1 Problem Formulation and Input Specifications

We start this section by formalizing the problem before we specify the input. In the next section, we begin to describe our system.

[5] Interestingly, the same is true for governor-dependent offset distance: the higher it is, the lower parsing accuracy. Our notion of path length ignores such distances. It may be interesting to investigate if they have an impact on argument classification performance.

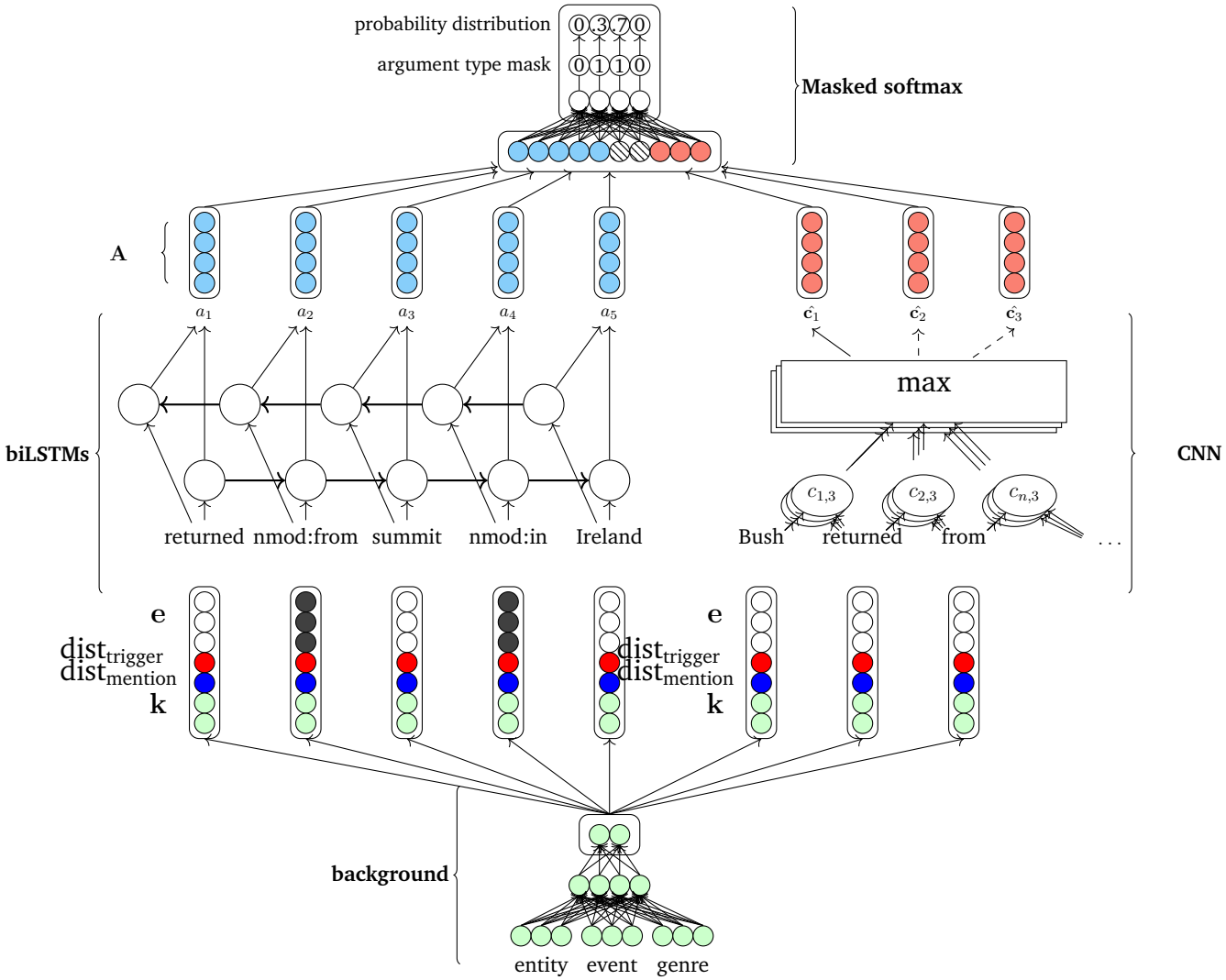


Figure 4.3: biLSTM/CNN architecture. Process flow is depicted from bottom to top. The background vector (**background**) is input to the bi-directional dependency path LSTM (**biLSTM**) and the lexical context CNN (**cnn**). Both produce representations, which are finally subject to a softmax distribution over argument types (**softmax**). Embeddings e depicted in white are fixed during training, every other node with learnable weights receives backpropagation updates. Section 4.3 describes each component in detail.

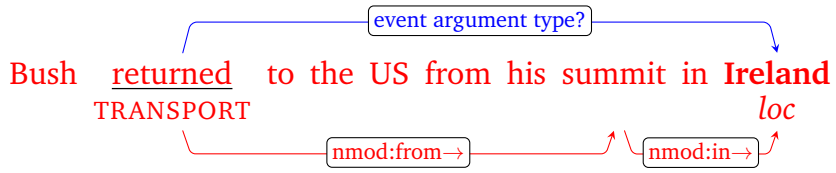


Figure 4.4: A training/test instance. Depicted in red is given information, depicted in blue is requested information. Given are (from top to bottom) the sentence, an event trigger and its event type, the entity type of the argument candidate, and the shortest dependency path connecting trigger and argument candidate.

Even though we call the task in this chapter argument *classification*, it is not inherently clear how it should be formulated. It could be stated as a structured prediction problem, for example (Chapter 3). However, in this chapter we want to focus on improving the prediction of individual arguments. To remove all influence from other factors than the actual argument as good as possible, we neither want our system to have features which interact with other argument decisions, nor do we want to have the problem of error propagation in any form, be it from other arguments or from trigger predictions. To avoid error propagation from wrong trigger decisions, we decided to divert from the traditional evaluation mode and to use gold triggers and therefore gold event types. To the best of our knowledge, we are the first to investigate argument classification prediction in isolation. We believe that our 'laboratory' conditions (especially gold triggers) are crucial to improve performance in this area, and to directly compare event argument performance. We will show in Chapter 5, that argument classification performance is strongly dependent on trigger classification recall.

For a trigger-entity mention pair (t, m) we make one (train/test) instance consisting of (a) event type, mention type, and text genre, (b) the shortest lexicalized dependency path $P_{t \to m}$ and (c) the sentence. Figure 4.4 visualizes such a pair ("returned", "Ireland"). Event type (TRANSPORT), entity type (*loc*) and genre (newswire, not depicted) constitute the first information group. The second group is the lexicalized dependency path (returned $\xrightarrow{\text{nmod:from}}$ summit $\xrightarrow{\text{nmod:in}}$ Ireland). The third group is the sentence. Given such information, the task is to predict a probability distribution over all argument types. Finally, we select the argument type with the highest probability.

The three groups described above correspond to the most valuable information sources of the baseline. However, the baseline draws only categorical features from them. Most notably, it relies on dependency paths seen during training because it cannot decompose them into meaningful subpaths, which is crucial for better classification performance (Section 4.2). The neural network architecture we present in

Section 4.3 is able to automatically construct more meaningful features by decomposing dependency paths and by learning embeddings for all information sources. In contrast to categorical features, where each feature has typically one weight, we learn more complex, multi-dimensional representations which enable the system to combine information on more than one scale.

We will now present the system architecture and formalize the three information groups and their respective component.

4.3.2 Background Vector

The background vector $\mathbf{k}$ provides a joint representation of the event type, entity type, and genre. The intuition behind the background vector is that arguments are expressed differently across event types, entity types, and genres; having a representation of the three information sources helps to capture these differences. $\mathbf{k}$ is input to the other two components and helps to learn better representations.

More formally, $\mathbf{k}$ is the last layer of a fully-connected three-layered feed-forward neural network[6] A linear layer n_i with input x is defined by

$$n_i(x) = \sigma\left(Wx + b\right), \tag{4.2}$$

where σ is a so-called non-linearity (typical choices are the tanh or sigmoid functions), W is a weight matrix, and b is a bias. Feed-wordward neural networks typically consist of multiple such layers. $\mathbf{k}$ in our case consists of three: input, hidden, and output, where the final layer is connected to all other parts of our system.

$\mathbf{k}$ is defined as

$$\mathbf{k} = \sigma(n_1(n_2(e(n) \oplus e(v) \oplus e(g))), \tag{4.3}$$

where σ is a non-linearity, n_1 and n_2 are two linear layers, e is an embedding function which assigns a different vector to each of its inputs, and n, v, and g are the entity type, event type, and genre. e assigns vectors drawing from a special random distribution (Glorot and Bengio, 2010) which is determined by the connections to the next layer, in our case to the input layer of $\mathbf{k}$.

[6] In other terms, this is a fully connected, feed-forward network with one hidden layer.

4.3.3 Representations of Lexicalized Dependency Paths

In this section we come to the core of our system, the component that produces representations of lexicalized dependency paths. In the following, we motivate, describe, and define all methods used for this component. We start with recurrent neural networks, and introduce the concepts of bi-directionality and especially Long Short-Term Memory networks afterwards.

4.3.3.1 Recurrent Neural Networks and Their Use

We observe that for our feature-based baseline argument classification performance drastically drops with the length of the dependency paths which connect trigger-argument pairs (Section 4.2). Our baseline encodes dependency paths as categorical features. It misses all paths which it has never seen during training. We hypothesize that a system which learns to capture similarities in dependency paths has a clear advantage because it can draw conclusions from dependency paths it never encountered during training; in other words, such a system can use dependency information more efficiently than feature-based systems like our baseline. Recurrent Neural Networks (RNNs) are particularly useful to learn (long-range) dependencies in sequences of varying length. They show good results in a variety of tasks similar to event extraction, such as relation extraction (Xu et al., 2015b) and semantic role labeling (Roth and Lapata, 2016).

We use RNNs to represent lexicalized dependency paths (Section 4.3.1). RNNs are trained to output high-dimensional vectors for input dependency paths, such that similar paths result in similar vectors. They offer the benefit that small variations in a path can either be ignored or completely change the overall representation, depending on the actual variation. For example, modifications like nmod:from may not change the overall 'meaning' of a dependency path; dobj and nsubj on the other hand sometimes encode antithetic roles like `Target` and `Attacker`. RNNs create a distributed (or continuous) representation of syntax paths, meaning that the path is processed elementwise and placed in a high-dimensional and latent space based on its elements. This stands in contrast to local (or discrete) representations of dependency paths which typically occur when they are used as categorical features for feature-based methods like logistic regressors or Support Vector Machines.[7] In other words, feature-based methods assign one weight to a path whereas RNNs embed the path in a latent, high-

[7] See Hinton et al. (1986) for a discussion of *local vs. distributed* representations.

dimensional space. Feature-based methods can only assign weights to paths which they encountered during training. RNNs can assign a vector to arbitrary paths, even if they were never encountered during training.

RNNs process a linear sequence 'through time'. For each time step i, they produce a state vector h_i (also called the hidden state), which is a combination of the input x_i, the previous state vector h_{i-1} and a bias b. Formally,

$$h_i = \sigma(Wx_i + Uh_{i-1} + b) \tag{4.4}$$

where W and U are weight matrices, and σ is a non-linearity, e.g, the tanh function. The hidden state h_i encodes information about the input at position i and about all previous time steps. In contrast to other deep learning methods, RNNs are able to produce a fixed-size representation for input of varying length.

We hypothesize that in many cases similar dependency paths lead to the same argument type. For example, attacked$\xrightarrow{\text{nsubj}}$US and attacked$\xrightarrow{\text{nsubj}}$Iraq are very similar and are both instances of `Attacker` arguments. Similarly, attacked$\xrightarrow{\text{nsubj}}$US and attacked$\xrightarrow{\text{nsubj}}US\xrightarrow{\text{appos}}$jets are similar and both instances of the same argument type. A change in grammatical relations however might lead to a different argument type. attacked$\xrightarrow{\text{dobj}}$Iraq for example indicates a `Target` instead of an `Attacker` because of the change from nsubj to dobj. The method we use to represent these paths needs to be able to capture both aspects – it needs to produce similar representations for paths which indicate the same argument type, and dissimilar representations for paths indicating different types. The same requirements hold for Semantic Role Labeling, where RNNs successfully improved performance. We believe that they improve argument classification performance as well. However, we do not put the entire modeling burden on our RNNs alone. Instead of using only the final representation vector they produce, we use every representation of every time step in the dependency path and let the higher classification layers decide about the relevance of individual segments.

4.3.3.2 Bi-directional Recurrent Neural Networks

RNNs encode information from previous time steps. In other words, they do not 'see' ahead. Only the last element contains information about the entire sequence. This is a drawback in our case because we use all the hidden states of the RNN for classification. Ideally, each of them would encode information about the entire sequence.

Bi-directional RNNs overcome the limitation of standard RNNs by processing their input from both directions – left to right, and right to left – and combining the representations afterwards. More formally, they produce two hidden states $\overrightarrow{h}_i$ and $\overleftarrow{h}_i$, and combine them via some composition function: $h_i = \odot(\overrightarrow{h}_i, \overleftarrow{h}_i)$. $\odot$ may be, for example, concatenation or averaging.

4.3.3.3 Bi-directional Long Short-Term Memory Networks

Unfortunately, standard RNNs are limited in their ability to learn long range dependencies because of the *vanishing or exploding gradients* (Bengio et al., 1994). These problems occur only during training and arise when using the (predominant) backpropagation algorithm: In order to backpropagate through RNN states, they are unfolded over time, forming a linear chain of hidden states, inputs, and outputs. The gradients are computed for the last time step, then for the second-to-last timestep with respect to the last, etc. Depending on the numerical values and activation functions involved, the gradients may either vanish (go towards zero) or explode (go towards infinity), the more time steps are considered. Exploding gradients lead to numerical overflows and effectively break training. Vanishing gradients have a more subtle effect: They greatly limit the learning ability of RNNs because errors have no influence on states further back in time.

LSTMs, short for Long Short-Term Memory Networks (Hochreiter and Schmidhuber, 1997), mitigate this problem by introducing the concept of *gates*. Gates control the information they let through. Coupled with an architecture which is able to carry information either changed or unchanged from one state to another, LSTMs effectively bypass the vanishing or exploding gradient problem. They are able to carry important information through the entire sequence, or block irrelevant information from processing. It becomes part of the learning problem which information is important or not for which states.

We will now present the formal definition of an LSTM and explain the gates afterwards. In the following, $\odot$ means the Hadamard product, a kind of matrix multiplication where the value a, b is the product of elements a, b in the original matrices.

$$i_j = \sigma(W_i^d x_j + U_i^d h_{j-1} + b_i^d), \tag{4.5}$$
$$u_j = \sigma(W_u^d x_j + U_u^d h_{j-1} + b_u^d), \tag{4.6}$$
$$f_j = \sigma(W_f^d x_j + U_f^d h_{j-1} + b_f^d), \tag{4.7}$$
$$o_j = \sigma(W_o^d x_j + U_o^d h_{j-1} + b_o^d), \tag{4.8}$$
$$c_j = i_j \odot u_j + f_j \odot c_{j-1}, \tag{4.9}$$
$$h_j = o_j \odot \tanh(c_j), \tag{4.10}$$

Note that the gate definitions in Equations 4.5 to 4.8 strongly resemble the definition of a simple RNN in Equation 4.4. Each gate has two weight matrices W and U, one for the input x_j, one for the previous state at $j-1$, and one bias b. The superscript d refers to the directionality (left or right). Effectively, one LSTM per direction is used. Finally, σ is the sigmoid function.

c_j (Eq. 4.9) is called the memory state. This state depends on the input gate i_j (Eq. 4.5) and the update gate u_j (Eq. 4.6) on the one hand, and the forget gate f_j (Eq. 4.7) and the previous memory state on the other. To build c_j, the LSTM can control how much of the current input it wants to consider (input gate), and how strongly (update gate). It can also control what to forget about the previous memory state (forget gate). Once c_j is built, the final hidden state h_j (Eq. 4.10) only depends on a transformation of c_j through the output gate o_j (Eq. 4.8).

Input to our LSTM is a lexicalized dependency path P as introduced in Section 4.2. Each element in P (words and dependency labels) is represented by a vector containing an embedding (either a word or a dependency label embedding), the lexical surface distance to the trigger and to the mention, and the background vector. More formally, for every element i in P we define

$$x(i) = e(i) \oplus \text{dist}_{\text{trigger}}(i) \oplus \text{dist}_{\text{mention}}(i) \oplus \mathbf{k}, \tag{4.11}$$

where $e(i)$ is either a pre-trained word embedding if i is a word or a randomly initialized dependency embedding otherwise. By randomly initialized we again mean that the initial values are randomly sampled from a special distribution (Glorot and Bengio, 2010) which depends on the number of connections to the next layers in our system. $\text{dist}_{\text{trigger}}(i)$ and $\text{dist}_{\text{mention}}(i)$ are also randomly initialized embeddings of lexi-

cal surface distances of an element to the trigger or the argument word, respectively[8], and $\mathbf{k}$ is our background vector as defined above.

P is processed by a bidirectional LSTM (biLSTM). As described in Section 4.3.3.2, using biLSTMs has the advantage that each hidden state contains information from the entire sequence; using only a forward LSTM limits the representations at each position to the left context. For a path element i, the biLSTM produces two hidden states $\overrightarrow{h_i}$ (by the forward LSTM) and $\overleftarrow{h_i}$ (by the backward LSTM). These vectors contain information about the respective input x_i, as well as the hidden states of previously processed elements, i.e. elements to the left of i for $\overrightarrow{h_i}$ and elements to the right for $\overleftarrow{h_i}$. We average the hidden states belonging to the same input vector to produce the final output of the biLSTM:

$$a_i = \frac{1}{2}\left(\overrightarrow{h_i} + \overleftarrow{h_i}\right) \tag{4.12}$$

One could combine the vectors differently, e.g., concatenating them. In fact, concatenation and averaging performed similarly in our experiments. We decided to average the vectors because concatenation doubles the representation size, whereas averaging keeps it constant.

The final outcome is a sequence A of averaged forward-backward representations of elements in P. Along with the output of our CNNs (Section 4.3.4), A is input to a fully connected linear layer which produces a probability distribution over valid argument types. The middle-left part of Figure 4.3 depicts the biLSTM and its final output A, the sequence of averaged forward-backward representation of P. LSTM cells are represented by circles. Inputs are either word embeddings (white dots) for all words like “returned”, “summit”, etc., or dependency embeddings (black dots) for the dependency labels “nmod:from” and “nmod:in”. Furthermore, we have embeddings for distances to the trigger and to the argument head word (blue and red dots), and our background vector (light green). The arrows which connect the circles with themselves and with their inputs and outputs visualize how LSTMs operate. For the forward LSTM, one circle is connected to the next. For the backward LSTM, the direction is exactly opposite. The hidden states of both are averaged (light blue dots above) and forwarded to final classification.

[8] We encode this similarly to, e.g., entity type embeddings where each entity type corresponds to one embedding vector. Here, each distance value corresponds to one embedding vector. Path elements representing dependencies have the same distance vector as the previous element.

4.3.4 Representations of Lexical Contexts

In contrast to RNNs, which are designed to capture the meaning of sequences, Convolutional Neural Networks (CNNs) are often used to produce bag-of-words-like representations. They were successfully applied to many NLP problems (Kim, 2014; Johnson and Zhang, 2015, inter alia).

CNNs are designed to recognize patterns in their input. In our case, the input is a sequence of word vectors, one vector for each word in the sentence. Our CNNs learn to recognize (lexical) patterns relevant for argument classification. For example, words like 'from' or 'out of' in the vicinity of an appropriate entity mention may indicate the presence of an `Origin` argument. Our CNN component captures information similar to categorical features like lexical contexts (words to the left and to the right of the argument candidate), except that we do not have a fixed window from which we collect them but regard the entire sentence as 'context'. As we will show in our evaluations (Section 4.4.1), CNNs contribute to the overall system a significant performance increase (1.4 micro-averaged F_1 points on average).

CNNs mainly consist of learnable weight matrices (called *filters* or *kernels*). CNNs compute the dot product (or: the convolution) of filters and subsequences of their input in a sliding window manner. The outcome of this dot product is higher, if the current input exhibits a pattern similar to what a filter learned to recognize.

The application of the same filter to different subsequences of the input makes the recognition more robust. It has the effect of abstracting from the exact position of a pattern to its mere presence. Furthermore, it typically leads to less parameters compared to LSTMs, and this in turn leads to better performance with less training data. However, CNNs only capture *local* regularities and dependencies.

A CNN can be defined by the number and size (width and height) of its filters, and by the offset difference of the sliding window (called the *stride*). A CNN with 100 filters of size $(2, 2)$ and stride 1 for example computes the dot product of 100 randomly initialized weight matrices of size $(2, 2)$ with the input $[x_0, x_1], [x_1, x_2], [x_n, x_{n+1}]$ ($x_i \in \mathbb{R}^2$), etc. Generally for the above CNN, a stride of *stride* leads to a convolution of $[x_n, x_{n+1}], [x_{n+stride}, x_{n+stride+1}]$, etc.

In image processing, filters can be easily understood and visualized: In the lower layers of a visual classifier, they typically learn to recognize simple features like edges, in higher regions they typically learn to recognize more complex features (eyes, car

tires, leaves, etc.). Here, the absolute position of features is of little interest, as long as their relative distance to other features is known.

In our case, filters learn patterns in sequences of word vectors. As for most NLP problems, our convolution height is equal to the dimensionality of the input vectors, mainly meaning that we do not learn patterns across embedding dimensions, only across embedding vectors.

We use CNNs to represent the lexical context of an event argument candidate. In our case, filters learns to recognize (lexical) patterns over sequences of word embeddings. We apply filters over the entire sentence (enriched with information about the lexical distance of a word the current trigger and argument candidate). A filter c at position i is defined as

$$c_i = \sigma\left(W x_{i:i+h-1} + b\right), \tag{4.13}$$

where σ is a non-linearity, W is a weight matrix (the filter), b is a bias, and $x_{i:i+h-1}$ is a subset the entire input sequence x of size h beginning at position i. The filter is applied as a sliding window of width h across x; the new position of the sliding window is defined by the tunable hyperparameter *stride*.

We apply max-pooling afterwards, i.e., the final output of one CNN filter is maximum value it produced after it processed the entire dependency path. Our CNN uses 50 filters for filter widths 2, 3, and 4. The middle-right part of Figure 4.3 exemplifies a CNN with 3 filters and filter width 2: Input to the CNN is a tokenized sentence where each word at position i is replaced by a vector $\mathbf{x}_i$, which is almost identical to the definition of $\mathbf{v}_i$ above – the only difference being that $\mathbf{x}_i$ contains only word embeddings (white dots). In the figure, x_1 for example consists of the word embedding for "Bush", x_2 for "returned", etc., both enriched with the distance embeddings and the background vector (red, blue, and light green dots, respectively). Filter are represented by ellipses; they are stacked for each position to indicate that we use multiple filters. For the same reason, they are also numbered: $c_{1,3}$ for example is filter number 3 applied to the first two inputs. Behind it is $c_{1,2}$, etc. Afterward all filters are applied to the entire sequence, we only keep the maximum values (indicated by the "max" rectangles). $\hat{c}_{1,1}$ (the first element in vector $\hat{c}_1$) for example is given by $\hat{c_{1,1}} = \max(c_{1,1}), c_{1,2}, c_{1,3})$. $\hat{c_1}$ has the same dimensionality as the number of filters – since we use CNNs with three different filter widths, we have three vectors $\hat{c}_1, \hat{c}_2, \hat{c}_3$ as the final output of the CNN

component. The last two are connected with dashed arrows because their filters and applications are not visualized in the figure.

4.3.5 Final Classification

Finally, the concatenation of the LSTM and CNN components outputs serve as input to a softmax layer which produces a probability distribution over all argument types (upper part in Figure 4.3). We pick the class with the highest probability as our final result.

However, choosing between *all* classes is unnecessary because not all combinations of event type, entity type and argument type are possible. For example, the argument type `Vehicle` can only be assigned to TRANSPORT events, and only mentions with entity type *veh* can be possible fillers. We modify softmax to assign zero probability to classes which are disallowed:

$$y_i = \frac{m_i e^{x_i}}{\sum_j m_j e^{x_j}} \tag{4.14}$$

The above equation gives the probability for a particular argument type, y_i, where $x \in \mathbb{R}^{29}$ is the input vector to softmax, and m is a binary vector indicating allowed types. m depends on the event type, because the possible roles also depend on them. ATTACK has a different m than `Be-born`, for example. Note that $y_i > 0$ only if the respective argument type is allowed.

We need the masked version because with standard softmax[9] we cannot fully control which argument types get zero (or near-zero) probability. The problem with standard softmax stems from the fact that $e^0 = 1$, meaning that we cannot zero out forbidden types; $x_i = 0$ would get a higher probability than $x_i < 0$, but we want the opposite: Values $x_i = 0$ should always have lowest probability. One possibility to circumvent this without modifying softmax is to set forbidden values to a large negative number, preferably to negative infinity. However, this is only possible if the framework allows computations with infinity; specifically, one would need the framework to compute $e^{-\infty} = 0$.

The top part of Figure 4.3 visualizes the softmax component. The input vector is first reduced to 29 dimensions (28 for each argument type, and one for $\emptyset$), multiplied with the restriction mask, and forwarded to our modified softmax.

[9] Standard softmax is defined as $y_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$, using the same symbols as above.

When we are in training, we can update parameter weights given the softmax distribution and a loss function.

4.3.6 Loss

We tried different loss functions and found *cross entropy* to work best. Formally, we minimize

$$\mathcal{L} = -\sum_{n}^{N}\sum_{i=0}^{k} y_{n,i}(x)\log(\hat{y}_{n,i}(x)), \tag{4.15}$$

where N is the sample size (number of training instances), k is the number of classes, and y and $\hat{y}$ are the true and the predicted probability of a specific class. We have a learning problem with multiple, mutually exclusive classes. In such a case, the cross entropy loss can be simplified, because $y_{n,i} = 1$ if class i is the correct class of sample n, and 0 otherwise. With this observation, we can rewrite Equation 4.15 to

$$\mathcal{L} = -\sum_{n}^{N}\log(\overline{y}_n), \tag{4.16}$$

where $\overline{y}_n$ is the *predicted probability of the true class* of sample n. This is 0 if true label probabilities for the entire training set are 1. Maximizing this loss corresponds to maximizing the true label probability for each training sample. Note that Equation 4.16 is also known as the *negative log likelihood loss*.

4.3.7 Training

We train over shuffled minibatches. During training, we keep word embeddings fixed, but dependency embeddings receive backpropagation updates. LSTMs are trained via *backpropagation through time* (Mozer, 1995). We use NADAM as the optimizer, an extension of ADAM (Kingma and Ba, 2014) with Nesterov momentum (Dozat, 2016).

We optimize hyper parameters for biLSTM/CNN using Random Search (Bergstra and Bengio, 2012) on the development set. Specifically, we can set the batch size, the learning rate, the bias vector and the LSTM hidden state dimensionalities, and the number of CNN filters. We found that the learning rate has a much lower effect on performance than the batch size and the weight for non-null training samples (explained

below). We always use a learning rate of 0.002. We use a batch size of 450, 20 CNN filters, 150 LSTM hidden state, and 130 background vector dimensions.

In order to deal with class imbalance, we set the weight of non-null training samples to 2; this value is used to scale the loss accordingly. Note that these weights have a direct impact on the gradients which are computed during backpropagation. In Section 5.6.2 we present a new undersampling method which has the same effect as increasing the weights of some classes without changing the gradients.

We used Keras (Chollet et al., 2015) version 2.0.2 with the TensorFlow backend as the learning framework. Training the network on an NVIDIA P40 GPU takes about 20 seconds per epoch.

4.3.8 Parameter Averaging

Inspired by the Averaged Perceptron (Freund and Shapire, 1999; Collins, 2002) we do not use the learned parameters θ directly for prediction. Instead, in each epoch we keep a moving average of the parameters:

$$\theta_\mu^T = \begin{cases} \alpha\theta^T + (1-\alpha)\theta_\mu^{T-1}, & \text{if } T > 1 \\ \theta^T, & \text{otherwise.} \end{cases}$$

θ_μ^T is the current averaged weight vector, θ^T is the current (non-averaged) weight vector *after* training epoch T, and θ_μ^{T-1} is the version of θ_μ from the previous epoch (after averaging weights for epoch $T-1$).

Please note that we do not use θ_μ for training, we use it only for prediction. We produce one averaged weight vector per epoch; for the final test set evaluation, we use those averaged weights which produced the highest development set F_1.

Using θ_μ instead of the non-averaged weights θ stabilizes training. More precisely, it reduces variance/overfitting by not allowing weights to oscillate unbounded and fit spurious characteristics of the training set. In this respect, it is similar to popular regularization techniques like $L1$ or $L2$; however, instead of pushing the weights towards zero, parameter averaging pushes them towards the average of all the versions encountered during training.

Figure 4.5 visualizes the effects of parameter averaging. Depicted are several training rounds with and without parameter averaging, represented by the green and blue areas, respectively. The areas are defined by min/max F_1 values against training

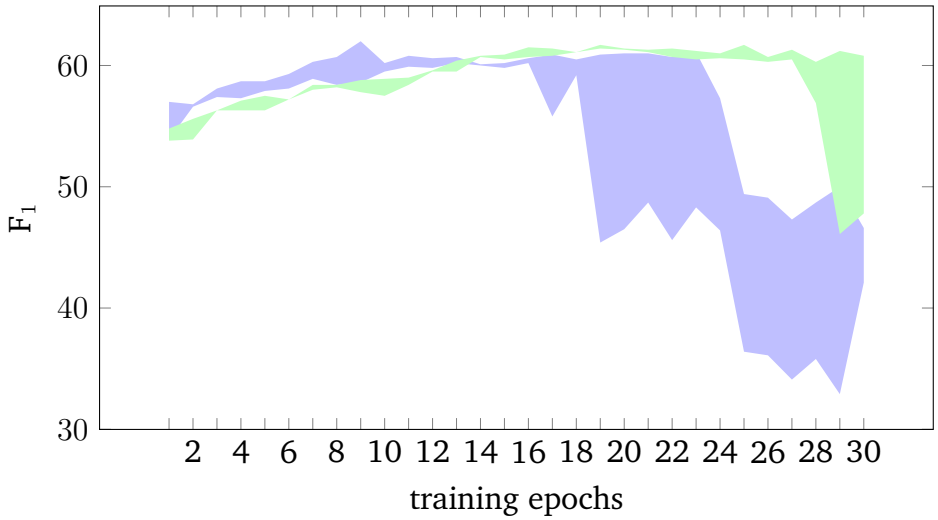


Figure 4.5: Development set F_1 against training epochs when training with (green) and without (blue) parameter averaging. Areas are defined by min/max F_1 values of several training runs.

epochs. The green area (parameter averaging) remains quite narrow across training epochs while the blue area diverges heavily, starting around epoch 18, indicating that there is a high variance. Furthermore, the green area indicates that training with parameter averaging maintains a better performance on the development set across multiple models and more training epochs, while training without parameter averaging produces worse results with a much higher variance more quickly. Both areas show the onset of overfitting; the blue area indicates that overfitting starts earlier without averaging (around epoch 20) than with averaging (around epoch 28). We can conclude that training with parameter averaging leads to better model generalization. Parameter averaging is largely ignored in the machine learning literature. We leave for to future research to investigate the realtions between parameter averaging, $L2$ regularization and dropout.

We have now defined biLSTM/CNN in terms of its architecture and training methods. The next section presents evaluation results.

4.4 Experiments and Results

Please refer to Section 2.2.6 for a description of the train-dev-test split we use here and to Section 2.4 for criteria when trigger and argument decisions are correct. We also adopt the proposal we make in Section 2.5.2 here: We always train and evaluate

five times, and report evaluation numbers averaged over these five runs. Furthermore, we report sample standard deviations.

As we already discussed in Section 4.3.1, we are interested in a feasibility study in this chapter – is it possible to enhance argument classification performance by learning to represent syntax structures? In order to evaluate this properly, we have to isolate argument classification from other related tasks. The task has two dependencies: entity mentions and event triggers. We must ensure that the systems we want to compare use the same entity mentions and event triggers, by ensuring that both are set to gold values. A drawback of this setting is that we have evaluation numbers only for biLSTM/CNN and our baseline, meaning that there is no direct comparability with previous work other than Li et al. (2013).[10]

Our baseline and biLSTM/CNN were both trained on the same training set using `enhanced++` dependencies (Schuster and Manning, 2016), and hyperparameters were optimized on the same development set. We follow standard evaluation procedures: An event argument is correct if its span and role match a reference argument (Ji and Grishman, 2008).

Training neural networks is usually a non-deterministic process. As we discussed in Section 2.5, it does not suffice to train one model and report one test set evaluation. In order to increase the reliability of our evaluations, training was performed five times, and test set evaluation was carried out for each of the five models. All numbers we mention in the following are *averages* of the five test set runs we performed. We additionally report *sample standard deviations* of all F_1 scores to give a more complete performance overview.

4.4.1 The Numbers

We report the results of four experiments. Specifically, we evaluate

- **Experiment 1**: micro-averaged performance
- **Experiment 2**: micro-averaged performance without context CNNs
- **Experiment 3**: performance per argument type
- **Experiment 4**: micro-averaged performance per path length.

[10] We discussed some possible influences of trigger detection on argument performance in Section 3.2.2.3 in more detail. Here, we must eliminate any such influence.

	Baseline			biLSTM/CNN				
	P	R	F_1	P	R	F_1	$\Delta F_1 \pm \sigma$	Support
Micro	67.7	58.7	62.9	63.1	68.2	65.5†	2.7 ± 0.5	916
no CNN				66.2	62.5	64.2†	1.3 ± 0.8	916
`Time`	69.9	70.9	70.4	70.9	80.1	75.2	4.8 ± 1.2	134
`Entity`	63.7	56.7	60.0	57.7	65.2	61.2	1.2 ± 2.9	127
`Place`	64.0	41.7	50.5	52.1	48.0	49.9	-0.6 ± 1.7	115
`Person`	74.6	61.7	67.6	69.1	78.3	73.4	5.8 ± 2.1	81
`Artifact`	78.5	71.8	75.0	70.3	77.2	73.5	-1.5 ± 1.2	71
`Destination`	63.4	66.7	65.0	65.6	80.0	72.1	7.1 ± 1.0	39
`Crime`	84.4	100.0	91.6	82.5	99.5	90.2	-1.3 ± 0.6	38
`Attacker`	60.7	47.2	53.1	52.4	66.6	58.6	5.5 ± 3.2	36
`Defendant`	70.0	63.6	66.7	67.6	75.2	71.1	4.4 ± 1.5	33
`Agent`	64.7	34.4	44.9	55.9	40.6	46.8	1.9 ± 6.8	32

Table 4.3: Test set argument classification precision, recall, and F_1 for the baseline and biLSTM/CNN, ordered by argument type frequency. Reported are argument types with more than 30 instances. biLSTM/CNN numbers are averaged over five test set runs. "**Micro**" and "**no CNN**" report micro-averaged numbers for Experiments 1 and 2, respectively; the other rows report numbers for Experiment 3. "$\Delta F_1 \pm \sigma$" reports the difference in F_1 between biLSTM/CNN and the baseline, as well as the respective sample standard deviation. "Support" reports the number of instances. † means statistically significant for all test runs at the $p < 0.05$ level. We measured significance only for micro-averaged numbers (Lines 1 and 2).

Experiment 1 is our main experiment – here, we evaluate how well biLSTM/CNN can predict event arguments compared to the baseline. We report micro-averaged evaluation measures, excluding $\emptyset$ (the 'null' class). Extensions of this are Experiments 3 and 4, where we report the same micro-averaged measures, but grouped per argument type and per dependency path length. We exclude lengths > 5 because of data sparsity. Experiment 4 compares biLSTM/CNN and the baseline not only against each other, but also performances for short vs. long paths. Finally, Experiment 2 checks the contribution of lexical context CNNs to the overall performance.

Table 4.3 reports evaluation numbers (precision, recall, F_1) for the baseline and biLSTM/CNN. Lines 1 and 2 correspond to Experiments 1 and 2, respectively, all other lines to Experiment 3.[11] Column "$\Delta F_1 \pm \sigma$" reports the difference in F_1 between biLSTM/CNN and the baseline – positive numbers meaning better biLSTM/CNN performance – as well as standard deviations.

[11] We omitted all argument types with less than 30 instances from Experiment 3. Appendix A.3 reports evaluation results for all argument types.

The structure of Table 4.3 is the following: Line 1 corresponds to Experiment 1, Line 2 to Experiment 2, and the following lines to Experiment 3. Experiment 4 is depicted in Figure 4.6 for better clarity.

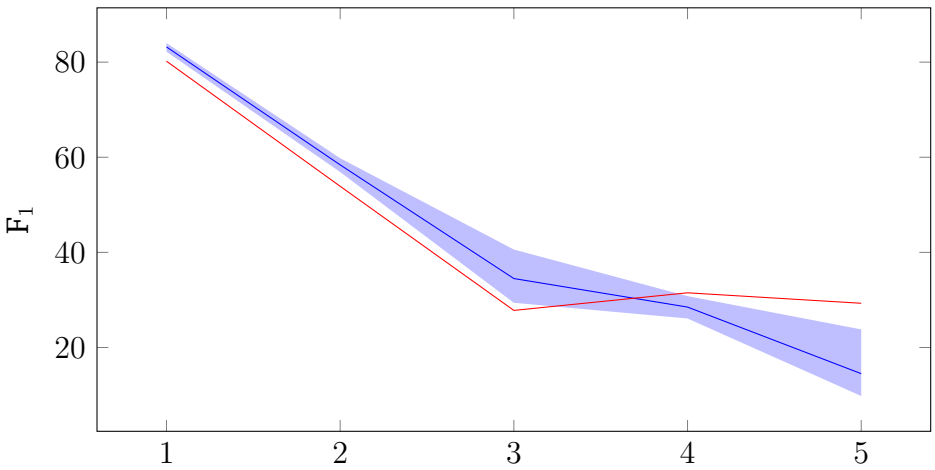
As we can see in Line 1, biLSTM/CNN has a lower precision and a considerably higher recall than the baseline, resulting in an increase of 2.7 points in micro-averaged F_1 (with a standard deviation of 0.5 F_1 points). This is statistically significant at the $p < 0.05$ level.[12] Note that biLSTM/CNN does not use any manually engineered features, whereas the baseline uses two dozen feature templates, resulting in 150,000 argument features. In contrast to the baseline, biLSTM/CNN also performs disjoint inference and cannot avoid certain error types (e.g., assigning the same role to two different entity mentions). The clear increase in argument classification performance is therefore encouraging.

When we compare performances without context CNNs (Line 2), we note that the system has a statistically significant improvement of 1.3 F_1 points over the baseline. However, compared to the full system, recall drops by 5.7 points, while increasing precision by 3.1 points, resulting in a decrease of 1.4 F_1 points. The main advantage of the CNN is that it makes the lexical context outside of the shortest dependency path available to the system, which reflects itself in the increased recall. However, a loss of 1.3 F_1 points on average is important – we can conclude that modeling the lexical context is important for the overall performance, even though lexicalized dependency paths contribute most to it.

When we look at individual argument types, we note that biLSTM/CNN improves performance for all but three types. `Destination` has the highest performance improvement (7.1 F_1 points), `Artifact` the highest loss (-1.5 F_1 points). `Time` as the most frequent type in the test data has a high improvement of 4.8 F_1 points. Standard deviations are low for most F_1 scores. However, they increase considerably for the lower half of the table where support falls under 40 samples – this reflects the uncertainty in the evaluation for infrequent argument types. Note that `Place` biLSTM/CNN F_1 is almost identical to baseline F_1; we can see from the standard deviation that biLSTM/CNN performance may well be above baseline performance if we would increase the evaluated number of models.

Figure 4.6 reports micro-averaged F_1 for the baseline and biLSTM/CNN per dependency path length (Experiment 4). Figure 4.6a is a visualization of Table 4.6b. In total,

[12] We measured significance using approximate randomization (Noreen, 1989). Each of the 5 models we trained performed significantly better than the baseline.



(a) Test set F_1 plotted against dependency path length. The red curve depicts baseline F_1, the blue curve depicts biLSTM/CNN F_1 averaged over five evaluations. The pale blue area reports min/max F_1 scores (best and worst scores for each path length across multiple models) for each path length.

	Baseline	biLSTM/CNN		
Length	F_1	F_1	$\Delta F_1 \pm \sigma$	Support
1	80.2	83.2	3.0 ± 0.7	432
2	53.9	58.4	4.5 ± 1.1	248
3	27.8	34.5	6.7 ± 4.1	123
4	31.5	28.5	-3.0 ± 1.8	59
5	29.3	14.5	-14.8 ± 6.2	26

(b) Test set F_1 by dependency path length for the baseline and biLSTM/CNN. "$\Delta F_1 \pm \sigma$" reports the difference in F_1 between biLSTM/CNN and the baseline, as well as the respective standard deviation. "Support" reports the respective number of instances.

Figure 4.6: Test set F_1 by dependency path length for the baseline and the averaged numbers of five biLSTM/CNN models (Experiment 4).

888 arguments (out of 916) were connected to their triggers by dependency paths of length 5 or less. biLSTM/CNN performs considerably better for lengths 1-3, especially for paths of length 2 (+4.5 F_1) and 3 (+6.7 F_1). Length-1 paths, which are nearly as frequent as all other path lenghts together, have an increased performance of 3.0 F_1 points. For length 4 and 5, biLSTM/CNN performance is lower than the baseline. However, there are few samples in these categories (59 and 26, respectively). The standard deviation for length-5 paths for example is very high (6.2).

The pale blue area in Figure 4.6a visualizes F_1 range: The lower bound consists of all min F_1 values for each path length, the upper bound of all max values. As we can see, the gap between min and max F_1 generally increases with path length; in other words, it generally increases as support decreases. biLSTM/CNN's entire performance interval lies well above the baseline for path lengths up to 3. These cases constitute 87.7% of all arguments in the test set. For path length 4, it is quite undecided: half of the times, performance is close to the baseline, half of the times, it is farther away. For length 5, biLSTM/CNN performance is well below baseline performance. However, there are only 59 and 26 length-4 and length-5 paths, respectively – these numbers seem too low to provide a reliable performance estimate.

4.4.2 Error Analysis

Figure 4.7 shows a confusion heat map for the predictions made on the ACE 2005 test set of a randomly selected model from the five models we used to produce the evaluation numbers above. The figure reports the number of classifications known to be in class i but predicted to be in class j. For example, there are 82 instances with gold and predicted label `Entity`, and 39 instances with gold label `Entity` but predicted label `Null`. The value $(\texttt{Null}, \texttt{Null})$ (in the figure's center) is truncated to the numerical value 100, every other point is unmodified.

There are two noticeable structures in Figure 4.7: The diagonal, meaning that in many cases the predictions are correct, and the cross shape, meaning that from all the errors made, the vast majority are confusions with the `Null` class, either in terms of false positives (wrongly assigned argument type) or false negatives (wrongly assigned null type). These cases constitute 93.3% of all errors (545 out of 584 cases). False positives and false negatives are comparably frequent: 48.5% out of all errors are false negatives, 44.8% are false positives. Form the remaining errors, `Entity` and `Place` are confused the most (6 times in total). In fact, `Entity` was only confused with `Place`.

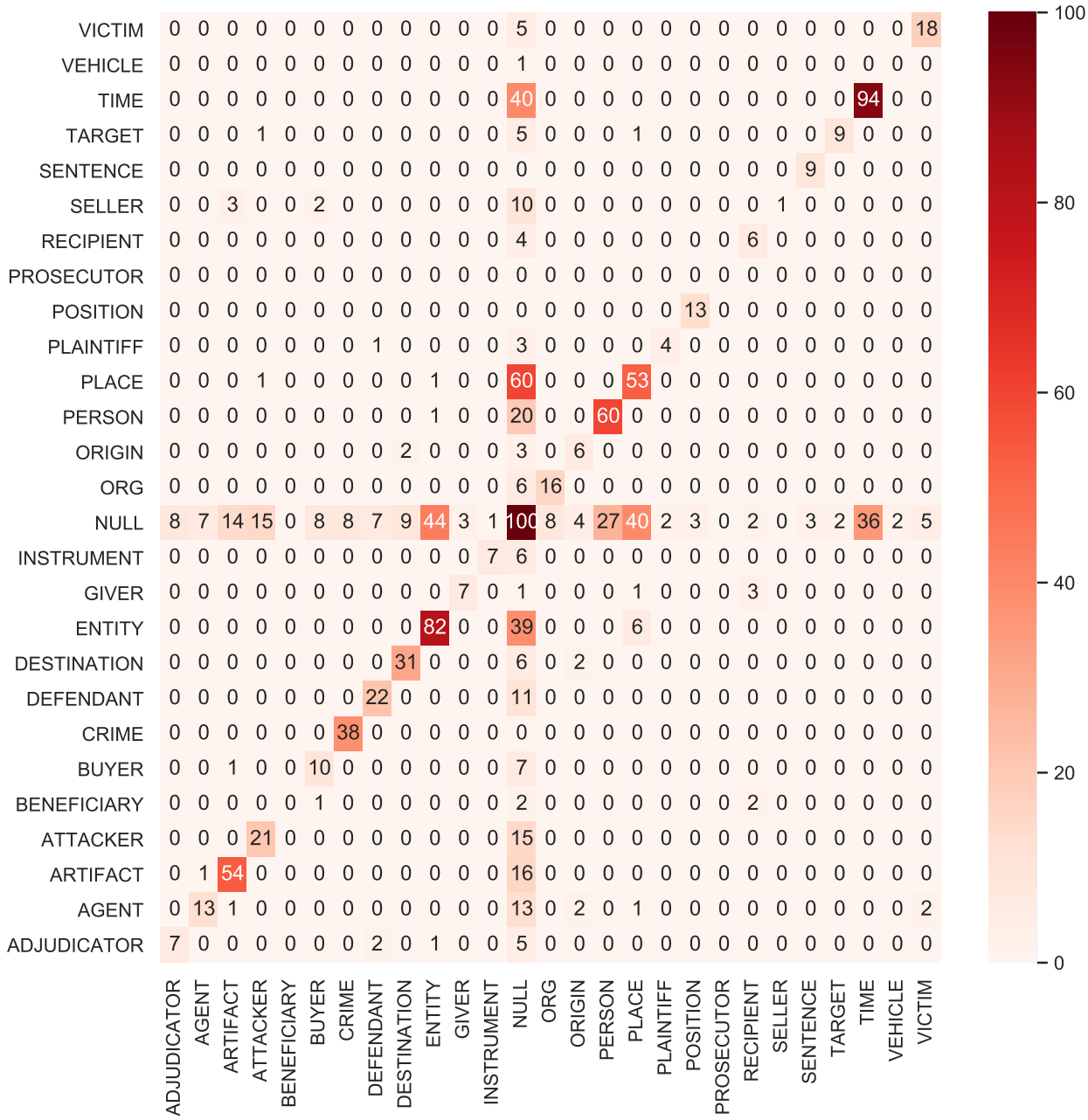


Figure 4.7: Confusion heat map (darker colors mean higher values) for biLSTM/CNN as measured on the ACE 2005 test set. A row-column value (i, j) represents the number of observations known to be in group i but were predicted to be in group j. The value at position $(\texttt{Null}, \texttt{Null})$ was truncated to 100 in order to make the relative differences between the other argument types clearer. All other values remained unchanged.

4.5 Conclusion

Event argument classification performance fluctuates quite heavily between different types – for the baseline, it is much harder to predict TARGET than VICTIM. We identified dependency path length as a main factor of this fluctuation. Based on this finding, and on the hypothesis that it is beneficial to be able to decompose long dependency paths in order to deal with paths never seen during training, we built a neural network which learns to represent shortest lexicalized dependency paths connecting triggers and potential arguments. We could show that such a system outperforms the baseline in terms of micro-averaged precision, recall,and F_1, in terms of individual argument types, and in terms of dependency paths up to length 3.

We eliminated the effects of trigger prediction on event argument classification by using gold triggers. In such a setting, we could prove that representing syntax is beneficial for the task. The next step is to research if syntax representations can help event extraction in general.

5 Syntax Encoding for Event Extraction

In Chapter 4, we show that distributed representations of syntax structures are useful for event argument classification. More specifically, we show that lexicalized shortest dependency paths help to better classify event arguments. However, in Chapter 4 we evaluate under laboratory conditions – when triggers are given. We therefore leave two major questions unanswered: (1) Will distributed syntax representations also benefit argument classification when triggers are predicted and noisy? (2) Do syntax representations also benefit trigger predictions?

The best way to answer both questions is to build a full event extractor which uses distributed syntax representations as a major information source. Most event extractors use syntax information in the form of categorical features or shallow local dependency relations. As we discuss in Chapter 4, categorical features have the disadvantage that they need to be encountered during training, otherwise a system cannot learn weights for them. Local dependency relations have the disadvantage that they may not bear enough information, especially for event argument classification.

The system and the methods we propose here address the two problems described above. Similar to Chapter 4, we strive for syntax representations which can assign meaning with respect to the event extraction task to arbitrary syntax structures and which are not limited to structures encountered during training. In contrast to Chapter 4 however we want to have representations suitable for the whole event extraction task, and these representations are necessarily more complex because, e.g., triggers may have more connections than one and thus require more than single, chain-like shortest dependency paths. We could encode multiple shortest dependency paths, one for each path connecting the trigger to one of the entity mentions in the sentence. We decided to unify such a representation and to encode the dependency graph of the sentence instead.

In this chapter, we explore different syntax encoders for event extraction. In particular, we analyze two methods in detail: Graph Convolutional Networks (Kipf and Welling, 2017; Marcheggiani and Titov, 2017) and tree-shaped LSTMs (Tai et al., 2015; Miwa and Bansal, 2016). A Graph Convolutional Network (GCN) encodes a graph node in terms its neighbor nodes. GCNs produce their encoding fast, but they only capture the immediate neighborhood of a node – neighbors of neighbors can only be encoded if two GCN layers are stacked. A tree-shaped LSTM (treeLSTM) on the other hand encodes the entire graph recursively, but need more computation time for the encoding, and typically more training data to perform well.

Our contributions in this chapter are the following.

1. We develop a modular event extractor which can be equipped with different syntax encoders. In its most basic form (without syntax encoders) this system is similar to the current state-of-the-art. We can plug in syntax encoders to the system and directly compare their performances with each other and the baseline, using the exact same preprocessing and infrastructure. This eliminates one of the most important sources of incomparability – different preprocessing.

2. We complete the work in Chapter 4 and investigate the use of distributed syntax representations for event extraction in general: Given the same preprocessing and the same underlying neural architecture, we compare the performance of different syntax encoders, namely Graph Convolutional Networks (Section 5.5.3) and tree-shaped Long Short-Term Neural Networks (Section 5.5.4).

3. We propose repeated negative undersampling, a method to increase trigger classification recall, and as a consequence also increase argument classification performance. We show that this undersampling strategy increases the performance of all methods across all data splits.

4. We show that training event extractors on the ACE 2005 data suffers from high variance. Therefore, we use *Bootsrap aggregating*, or *bagging* (Breiman, 1996) for event extraction. Bagging leverages the high variance of our training data by randomly sampling different versions of it, and training a classifier on each sample. Classifiers form an ensemble which is then used to predict triggers and arguments. The ensemble shows a smaller variance across different training runs and reliably increases performance.

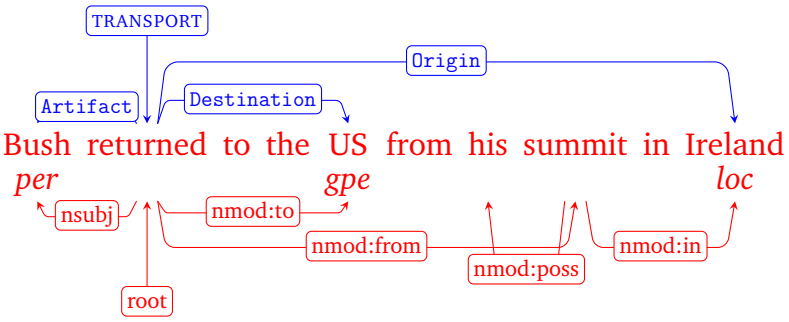


Figure 5.1: Input (red) and output (blue) for EVENTOR. The sentence, the corresponding enhanced++ dependency parse, and all entity mentions are input. Output is the entire event structure, i.e., all trigger and argument assignments. Note that a particular entity mention can bear multiple argument labels if it is argument to multiple events.

5. We evaluate not only on the widely used data split, but also on two other random splits which follow the distribution of the entire ACE 2005 data set more closely. We train and evaluate multiple models for each new split and gain a more reliable evaluation than the usual evaluation standard reported in the literature.

This chapter is structured as follows. Section 5.1 formulates the problem. Sections 5.2, 5.3, and 5.4 describe the general architecture of the system, how the context, and how categorical features are encoded, respectively. Section 5.5 formalizes the main methodological part of this chpater: the syntax encoders. Section 5.6 describes details of the training procedure, including *repeated negative undersampling* and *bagging*. Finally, Sections 5.7 and 5.8 describe the experiments and results in detail, and give the conclusion, respectively.

5.1 Problem Formulation

Before we present the actual system, we want to define inputs and outputs. Figure 5.1 visualizes them. All inputs are in red: the sentence ("Bush returned to the US from his summit in Ireland"), its enhanced++ dependency parse (Schuster and Manning, 2016), and all entity mentions ($[Bush]_{\text{PER}}$, $[US]_{\text{GPE}}$, and $[Ireland]_{\text{LOC}}$). Given all this information, we want to predict the full event structure, including all triggers and the respective arguments.

Our main contribution is the focus on the syntactic structure. The syntax encoders we present in the following can place the entire dependency graph in a high-dimensional

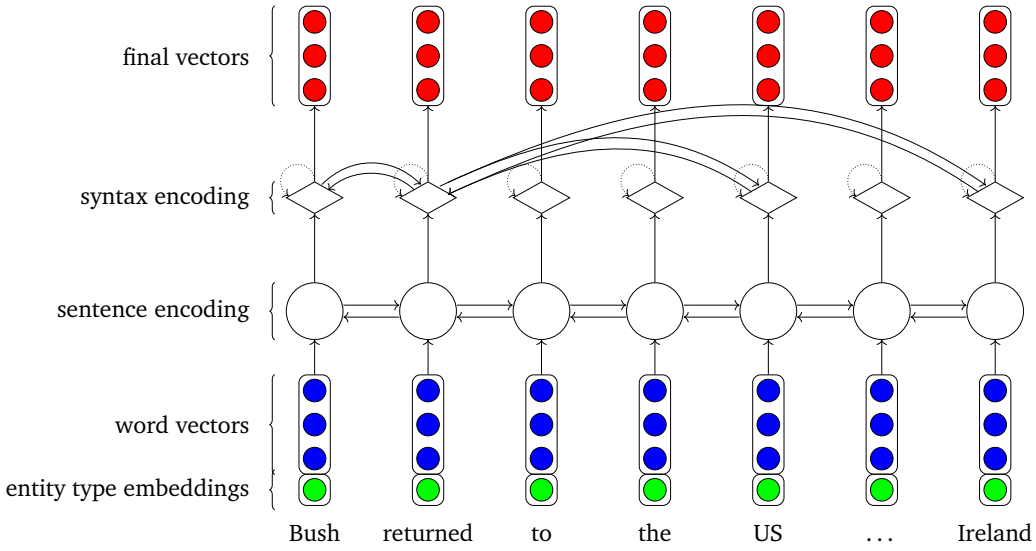
vector space such that similar graphs are close in the space. Furthermore, we can exchange the syntax encoders or leave them out within the same system, using the same infrastructure and the same preprocessing. This gives us the possibility to directly and reliably compare the syntax encoding methods we want to investigate, and show their use for event extraction. We will now describe our system (EVENTOR).

5.2 System Architecture

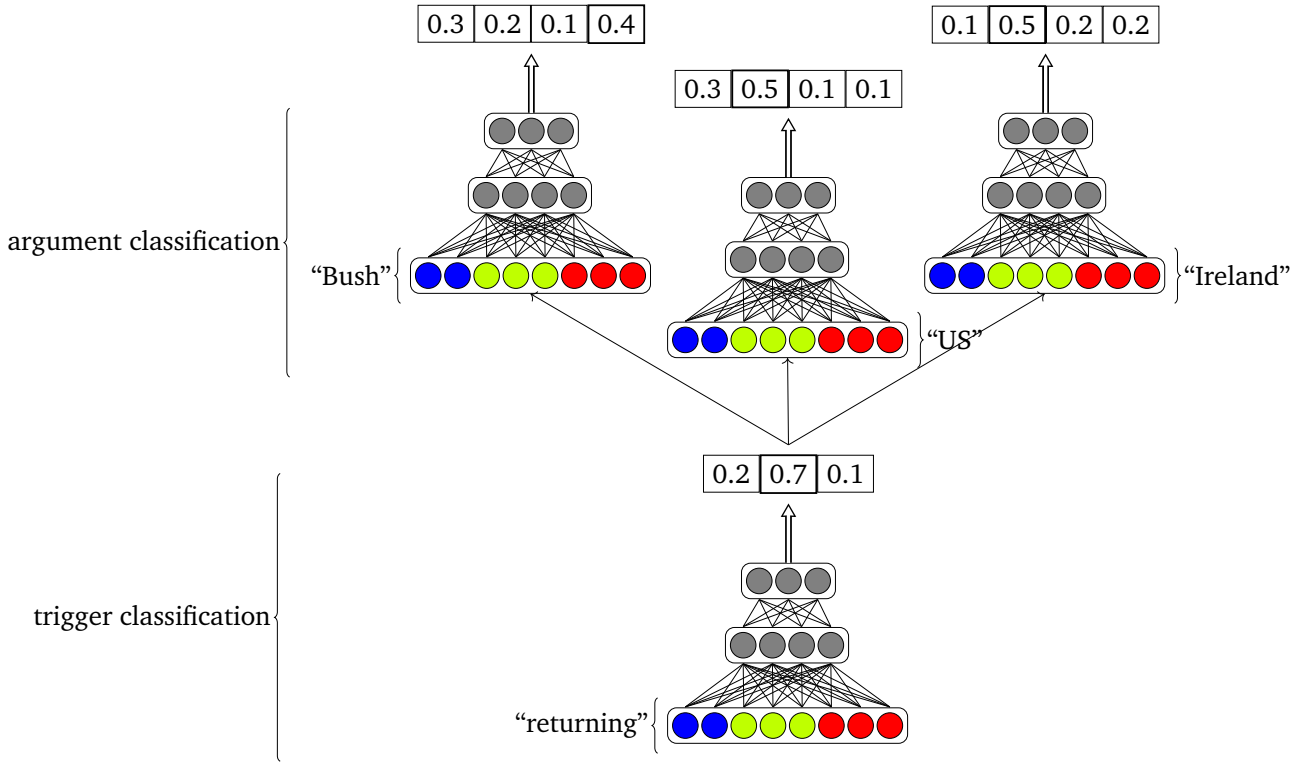
EVENTOR is composed of two encoding layers and two classifiers, one for triggers, one for arguments. The first encoding layer produces a *sentence encoding*. The second layer produces a *syntax encoding*. The syntax encoding, along with categorical features and context word vectors, is used as input for the trigger and argument classifiers. If the trigger classifier predicts that a word is a trigger, all entity mentions in the sentence are subject to argument classification with respect to this trigger.

Figure 5.2a visualizes the system architecture. Processing flow is depicted from bottom to top. Word vectors (blue) and entity type embeddings (green) are input to the sentence encoder, a bi-directional LSTM (Section 4.3.3.2), represented by circles and left-ro-right arrows. Entity embeddings are randomly initialized and learned during training. The sentence encoding is input to the syntax encoder, represented by diamonds and bi-directional edges between dependents and governors. Dotted self-connections are needed for Graph Convolutional Networks, but omitted otherwise.

Once we produce the syntax encoding, we are ready to predict triggers and arguments. Figure 5.2b illustrates the process. The lower part ('trigger classification') has as input the context word vectors (Section 5.3, blue), a feature hash (Section 5.4, lime), and the syntax encoding (Section 5.5, red). The concatenated vectors are then input to a feed-forward neural network with one hidden layer. Finally, a probability distribution over event types (including $\emptyset$) is computed. If an event is predicted, the upper part of the figure is triggered ('argument classification'). Here, we iterate through every entity mention in the sentence and compute its argument type (including $\emptyset$). Input to the argument classifier is the same type of vector as above (concatenation of context words vectors, feature hash, and syntax encoding). For multi-word entity mentions we let the last word represent the entirety. Please note that we again follow the standard setting in event extraction and make use of gold entity mentions.



(a) EVENTOR's encoding layers. Green dots are entity type embeddings, blue dots are word embeddings. Circles represent LSTM cells, diamonds syntax encoder cells. For LSTMs, each word is connected to the previous and next word. For syntax encoding, each word is connected to its dependents and governors.



(b) EVENTOR final classification layer. Red dots represent the output of the syntax encoder. Lime dots represent categorical features. Blue dots represent context word vectors. The final output of each classification step is a probability distribution over trigger or argument types, respectively.

Figure 5.2: EVENTOR system architecture. The lower part visualizes the sentence and syntax encoders, the upper part the trigger and argument classification. Argument classification is only executed if the trigger classificator predicted an event type.

We will now formalize the system architecture from least to most complex, starting with context and categorical feature vectors, before formalizing the sentence and syntax encoders.

5.3 Context Vectors

The least complex information type we use for trigger and argument classification is context word vectors. Following Chen et al. (2015) and Nguyen et al. (2016), we use word vectors corresponding to the words left and right of word i (the word of interest), as well as the word vector of word i itself as input for the trigger and argument classifiers. Formally, we produce a context vector

$$c = v_{i-n} \oplus v_{i-n+1} \oplus \ldots \oplus v_i \oplus v_{i+1} \oplus \ldots v_{i+n}, \tag{5.1}$$

where v_x is the word vector of word x, n is the context window size, and $\oplus$ is the concatenation operator. Note that c is not input to the sentence or the syntax encoder described below, it only contributes to the final trigger and argument classifiers.

5.4 Categorical Features

Li et al. (2013) developed a predictive and diverse feature set for event extraction. Nguyen et al. (2016) and Zhang et al. (2017) show that these features also benefit deep learning systems. We follow them by incorporating argument classification features into our system. The 'argument' part of Table A.1 reports all features which are used here. Note that this feature set is identical to the static features used by the base system in Chapter 3.

The feature set size is in the order of 100,000 features. To keep classification tractable, we employ feature hashing (Weinberger et al., 2009). The idea is simple – instead of using the entire feature vector with 100,000 dimensions, we hash each feature string (i.e., produce a number for the string) and map it to a much smaller numerical range than the dimensionality of the initial vector.

Suppose we have a categorical feature vector $\mathcal{F} \in \mathbb{R}^m$, where $m \approx 100,000$. Neural networks usually use such a categorical feature vector in the form of a binary encoding, $\mathcal{B} \in \mathbb{B}^m$, where $\mathcal{B}(\xi(f)) = 1$ if feature f applies. ξ is a function $\xi : \mathcal{F} \rightarrow \mathbb{N}, f \mapsto x$ which transforms a feature f into an index x. When $\mathcal{B}$ is used in a neural network, it is

usually multiplied with many hundreds of weights, resulting in a great computational overhead. The feature hashing trick replaces ξ with a hashing function $\xi' : \mathcal{F} \rightarrow [0, n]$, with $n \ll m$. In this way, the feature vectors are much smaller and computation equivalently faster. The drawback of this method is that sometimes distinct and probably unrelated features are mapped to the same index (*feature collision*). Weinberger et al. (2009) prove for many tasks that collisions have no great impact on performance. We use *MurmurHash3* (Wikipedia contributors, 2018) with a liner transformation $\xi(x) \mod n$ as ξ'.

5.5 Encoders – From Word Vectors to Syntax Representations

In the following sections, we define and explain our encoders. We employ two types: The *sentence encoder* (Section (5.5.1)) produces a representation of the sentence and forwards it to the *syntax encoder* (Section 5.5.2), which adds a representation of the dependency structure. The sentence encoder operates on words and entity mentions, the syntax encoder operates on dependency relations. Please note that whenever we write that a vector or matrix is randomly initialized, we refer to a normal random initialization Glorot and Bengio (2010). We will now describe both in detail, with a focus on syntax encoders.

5.5.1 Sentence Encoder – Forming Word Representations

Input to the sentence encoder is a vector sequence. Each vector is the concatenation of a word and an entity type embedding. Formally, for a sentence of length n,

$$I = (x_0, x_1, \ldots, x_n), \tag{5.2}$$

where I is the input sequence and

$$x_i = v_i \oplus e_i. \tag{5.3}$$

v_i is a word embedding, e_i is an entity type embedding for the word at position i. Word embeddings are pre-trained and not updated, entity type embeddings are randomly initialized and updated during training. $\oplus$ is the concatenation operator.

Each word in the sentence is assigned an entity type label using the BILOU scheme (Ratinov and Roth, 2009): The first word of a multi-word entity mention receives the label 'B' (begin), the last 'L', and the others 'I' (inside). Single-word mentions receive the label 'U' (unity). If a word does not belong to any entity mention, it receives the label 'O' (outside). These labels are paired with the respective entity type: 'B-PER', 'U-FAC', etc. If multiple entities span one token, all applicable labels are sorted and joined to form a single label, e.g., 'B-NUMERIC-I-SENTENCE'.

The sentence encoder produces the representation of a word in terms of its word vector, entity type label, and the word sequence before and after it. Input to the sentence encoder is I, as defined above. The actual encoder is a bi-directional LSTM (Section 4.3.3.3). The sentence encoder produces a sentence encoding H or for input I, where

$$H = (h_1, h_2, \ldots, h_n) = \phi(I). \tag{5.4}$$

ϕ is a bi-directional LSTM. Each h_i is the concatenation of the forward and backward output of the bi-directional LSTM, $h_i = \overrightarrow{h_i} \oplus \overleftarrow{h_i}$. Note that an LSTM hidden state h_i belongs to the word at position i.

5.5.2 Syntax Encoder – Forming Syntax Representations

The sentence encoding H is input to the syntax encoder. The task of the syntax encoder is to link syntactically related words which may otherwise we far apart; furthermore, it also provides the dependency relation (nsubj, dobj, etc.) between the words. In Chapter 4, we show that syntax representations benefit argument classification. Here, we follow the same argumentation – syntax representations provide a condensed view on word relations, they bring the words which are most important to event extraction and which may be expressed far apart in a sentence closer together. Consider again the input and output visualization in Figure 5.1. The lower part shows the enhanced++ dependency graph, the upper part the event structure. It is immediately apparent that one is mirrored in the other. For example, the subject of the trigger word ("returned") is also the `Artifact` of the event, and its 'to'-modifier is the `Destination`.

In Chapter 4 however, we investigated the use of syntax representations for argument classification. Here, want to investigate its use for event extraction as a whole. Our intention is twofold: Can the finding from Chapter 4 also hold with predicted and noisy triggers, and can syntax representations help to improve trigger classification?

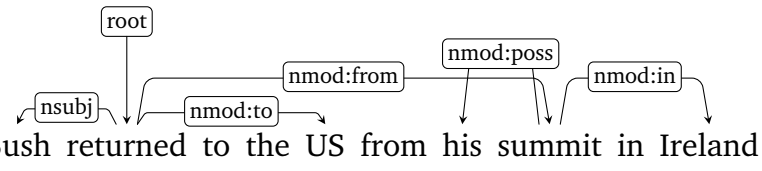
Our system from Chapter 4 (biLSTM/CNN) uses bi-directional LSTMs to encode lexicalized dependency paths connecting triggers and arguments. There, we encode a one-to-one relation between a trigger and an argument. Here, we want to model a one-to-many relation between one trigger and all argument candidates in the sentence. The representations we build are therefore more complex than in Chapter 4. On an abstract level, we go from representing linear chains to representing non-linear graphs. On a more concrete level, we go from representing lexicalized dependency paths to lexicalized dependency graphs. We call the representation of each node in the dependency graph a *syntax encoding*, and the component producing the representation a *syntax encoder*.

More formally, the syntax encoder produces a representation $S = (s_1, s_2, \ldots, s_n) = \theta(H, D)$ for the sentence representation H and a (directed, cyclic, labeled) dependency graph $D = (V, E)$ with V as the words in the sentence and E as dependency relations between these words. Words which do not participate in any enhanced++ dependency (e.g., prepositions) are omitted from all computations in the syntax encoder; they are, however, captured by the sentence encoder.

We decided to use two different syntax encoders which differ in scope: On the one hand, we use Graph Convolutional Networks (Kipf and Welling, 2017; Marcheggiani and Titov, 2017) which produce a syntax encoding based on the input and on the immediate neighbors of the current node. On the other hand, we use treeLSTMs (Tai et al., 2015; Miwa and Bansal, 2016) which produce an encoding based on the entire minimum spanning tree of the dependency graph. In the following, we will qualify the dependency graph D and its attributes before we present our syntax encoders.

5.5.2.1 D is directed

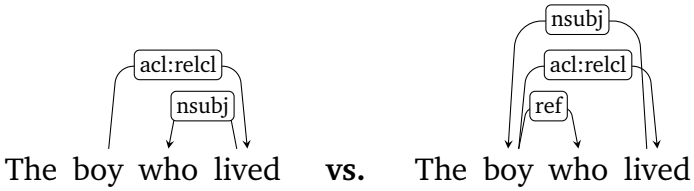
D is directed because dependencies distinguish between governor and dependent. This poses an important design problem for computation – should the direction be considered or discarded? We want to exemplify some of the implications related to directionality on the dependency graph of our example sentence, depicted in more readable form below:



If θ, the syntax encoder, respects directionality directly in the form of computational direction, it misses syntax information when it encodes words in dependent relations. In other words, it cannot look back: When θ encodes "returned" for example it sees "Bush" as the subject, but when in encodes "Bush", it misses that the word is the subject of "returned". To make the encoding of "Bush" also see "returned", we can only transform D to an undirected graph. We can, however, regain the directionality information by introducing an extended, direction-sensitive set of dependency labels. For the nsubj relation between "Bush" and "returned", we add two new labels, $\leftarrow$ nsubj and $\rightarrow$ nsubj, which point in the direction of the governor. The syntax representation for "Bush" encodes the relation Bush$\xrightarrow{\text{nsubj}}$returned; the representation for "returned" on the other hand includes (among others) the relation returned$\xleftarrow{\text{nsubj}}$Bush.

5.5.2.2 D is cyclic

Because we use enhanced++ dependencies (Schuster and Manning, 2016), D is cyclic. Cycles are introduced by relative clauses. Enhanced dependencies add the dependency 'ref' to a word and its relative pronoun, and the governor of the relative clause is attached to the predicate of the clause (Enhanced Dependencies Website, 2017):



Local syntax encoders deal with cycles naturally because they produce an encoding before they move on to encode neighbors of the current node. A Graph Convolutional Network for example captures the connections boy$\xrightarrow{\text{acl:relcl}}$lived and boy$\xleftarrow{\text{nsubj}}$lived when it produces the representation of "boy". Recursive methods like treeLSTMs however cannot handle cycles because they follow an entire graph before they produce an encoding. This means that they fall into an infinite loop if the graph is cyclic.

We resolve cycles by computing a minimum spanning tree with regard to root node distance. The root node in our example is "lived". If a recursive method follows the enhanced dependency graph, it reaches "boy" via the nsubj relation first – the minimum spanning tree does not contain the reverse edge connecting "boy" to "lived" via the acl:relcl relation, because it only keeps the edge it encounters first.

Please note that the minimum spanning tree is input to all syntax encoders. This means that we transform the dependency graph to a dependency tree before be produce any syntax representations. We decided to always use the minimum spanning tree instead of the graph to ensure all methods operate on the same structure.

5.5.2.3 D is labeled

When encoding dependency graphs, we do not only have to consider the graph structure, but also the edge labels (dependencies). There are at least three intuitive methods to incorporate syntax labels into neural networks: binary dependency vectors, embeddings or label-dependent weight tensors. In the following, we will use d as the number of dependency labels.

A *binary dependency vector* has d elements, one for each direction-sensitive dependency label ($\xleftarrow{\text{nsubj}}$, $\xrightarrow{\text{nsubj}}$, $\xleftarrow{\text{dobj}}$ etc.). Each dimension either contains a 1 or 0, meaning that the corresponding dependency label is present or not, respectively. On the one hand, such a vector is compact, on the other it only provides the information that a dependency label is present. The representations we describe below associate more information with each label which can be leveraged by the trigger and argument classifiers. In all our experiments, binary dependency vectors gave no measurable performance increase. We suspect that they do not contain enough information for event extraction on the ACE data.[1]

Dependency embeddings are similar to word embeddings. Each dependency label corresponds to some embedding vector; the totality of these vectors forms a $m \times d$ matrix where m is the representation dimensionality and d the size of the direction-sensitive dependency label vocabulary. This matrix is randomly initialized and receives backpropagation updates during training, meaning that the system learns dependency embeddings in a way useful for the task at hand. In contrast to dependency vectors, dependency embeddings can assign a 'meaning' to labels and not only indicate their presence. In contrast to label-dependent weight tensors (see below), they need considerably less training data to produce meaningful representations. In our experiments, dependency embeddings proved better than the other two representation forms. Both, our GCNs and treeLSTMs will make use of such embeddings.

Label-dependent weight tensors refer to the possibility that weight matrices are conditioned on dependency labels. The treeLSTM for example could have a child weight

[1] They are however used in Nguyen et al. (2016).

matrix for the nsubj relation, one for the dobj relation, etc. Conditioning weights on dependency labels gives the finest level of dependency encoding – it increases the number of weights drastically, however, because all relevant weight matrices need to be copied d times. Such tensors can be prohibitively large, at least for GPU training. Furthermore, they need a much higher amount of training data to produce reliable results compared to the other two methods described above.

5.5.3 Syntax Encoder: θ_{GCN}

The first syntax encoder we explore are *Graph Convolutional Networks*, or GCNs. We will denote this encoder frequently by θ_{GCN}. Graph convolution was successfully applied to a variety of tasks similar to event extraction, e.g., Semantic Role Labeling (Marcheggiani and Titov, 2017) or document classification on citation networks (Kipf and Welling, 2017).

The key feature of GCNs is that they encode a graph node in terms of its input features and local neighborhood. In this respect, it strongly resembles collective classification algorithms (Section 3.2.3). In fact, Kipf and Welling (2017) use a collective classification algorithm (ICA) as a comparative baseline for their GCNs. They find that GCNs outperform ICA across four datasets and two tasks.

GCNs produce the syntax representation s_i of a node i based on its neighbors $\mathcal{N}(i)$ and its input h_i. Formally,

$$s_i = \sigma\left(\sum_{n \in \mathcal{N}(i)} (W h_n + b)\right), \tag{5.5}$$

where σ is a non-linearity, $\mathcal{N}$ is a function which enumerates all neighbors of a node, $W \in \mathbb{R}^{l \times m}$ is a weight matrix, $h_n \in \mathbb{R}^m$ is the input vector, $b \in \mathbb{R}^l$ is a bias, l is the representation dimensionality, and m the input dimensionality. Note that $i \in \mathcal{N}(i)$; otherwise, the input vector h_i of node i would not affect its own representation.

From Equation 5.5 we can see that the representation of a node only depends on the representations of its neighbors and its own representation (because $i \in \mathcal{N}(i)$). If we want to encode the information provided by neighbors of neighbors, we have to perform a second GCN encoding pass, whose input is the output of the first pass. Similar to Marcheggiani and Titov (2017) we find no benefit in stacking GCN layers for event extraction. We follow their argumentation and suspect that our sentence encoder, which is input to the GCN layer, already provides 'global' information to some

extent; at least to an extent which makes additional, higher-order GCN encodings redundant. This might change with more training data.

In order to incorporate dependency labels, we can parametrize the GCN weight matrix W by labels, resulting in a label-dependent weight tensor, or we could parametrize the bias b by labels, resulting in a dependency embeddings matrix.

Marcheggiani and Titov (2017) find in their experiments that a hybrid method works best: Instead of parametrizing the weight matrix by dependency labels, they parametrize it by dependency directions (left, right, self-loop).[2] Additionally, they parametrize the bias by dependency labels. In our initial experiment, this model version did not perform better than one where only the bias is parametrized by dependency labels (making it equivalent to dependency embeddings, Section 5.5.2.3). Formally,

$$s_i = \sigma\left(\sum_{n \in \mathcal{N}(i)} \left(W s_n + b_{\text{dep}(i,n)}\right)\right), \tag{5.6}$$

where dep(i, n) is a function specifying the dependency between nodes i and n. $W \in \mathbb{R}^{l \times m}$ is the weight tensor, and $b_{\text{dep}(i,n)} \in \mathbb{R}^{l \times d}$ is the bias. l is the representation dimensionality, m the input dimensionality, and d the number of dependency labels. Please note that $i \in \mathcal{N}(i)$ and $s_i^0 = h_i$. Finally, σ is a non-linearity. Leaky ReLU (Xu et al., 2015a) gives the best results in our experiments.

The standard GCN as defined in Kipf and Welling (2017) gives uniform importance to each edge. However, this is usually not desired for NLP applications, especially when processing dependency graphs. On the one hand, dependency edges are not equally important for every task. E.g., a nsubj edge is presumably more important for event argument identification than a det edge. On the other hand, parsers do not produce perfect output. By weighting edges differently, the system may be able to compensate prediction weaknesses of the syntax parser. To introduce edge weights, we follow Marcheggiani and Titov (2017) and use scalar gates:

$$g_{i,n} = \text{sigmoid}\left(s_i \cdot \hat{v} + \hat{b}_{\text{dep}(i,n)}\right), \tag{5.7}$$

[2] Remember that we transform the directed dependency graph to an undirected version by introducing a 'reversed version' of each dependency edge with switched governor and dependent, and with a direction marker.

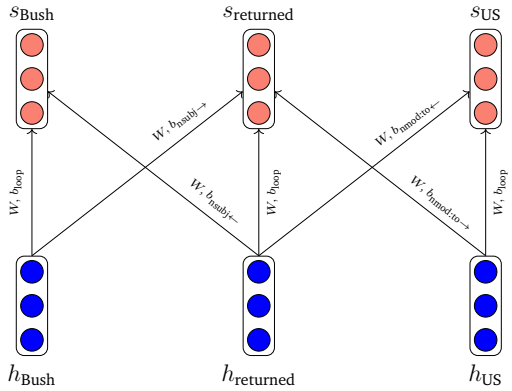


Figure 5.3: 1st-order Graph Convolutional Network. Processing direction is from bottom to top. Input to the GCN layer is H, the sentence representation. Edges are labeled with the exact weights and biases involved in the computations. Gates are omitted from visualization.

where $\hat{v} \in \mathbb{R}^m$ is a weight vector and $b_{\text{dep}(i,n)} \in \mathcal{R}^d$ is a bias. Again, the bias is parametrized by dependency labels. $g_{i,n}$ is a scalar representing the importance of the corresponding edge.[3] The final GCN state of node i is then defined by

$$s_i = \sigma\left(\sum_{n \in \mathcal{N}(i)} g_{i,n}\left(W s_i + b_{\text{dep}(i,n)}\right)\right). \tag{5.8}$$

Figure 5.3 visualizes a 1st-order GCN. Process flow is depicted from bottom to top. Input to the GCN layer is H, the sentence representation. Output is S_{GCN}, the first-order syntax representation of the dependency graph, consisting of the syntax representation s_{Bush} for "Bush", etc. Edges are labeled with the exact weights and biases involved in the computation. We omit all gates from the figure for brevity.

As we can see, s_{Bush} is influenced by h_{Bush} via a self-loop (we add the respective label to our dependency set) and by h_{returned} through the nsubj$\leftarrow$ relation and the respective weights, especially the respective bias weights corresponding to this relation.

[3] In their GitHub repository (`https://github.com/diegma/neural-dep-srl/blob/master/nnet/models/GraphConvolutionalLayer.py`), Marcheggiani and Titov (2017) compute a softmax probability distribution over all edges instead of using gates as defined in their paper. It is worth to evaluate gating against softmax distributions in future work.

5.5.4 Syntax Encoder: $\theta_{\textbf{treeLSTM}}$

The second syntax encoder we explore is an extension of the *tree-shaped Long Short-Term Memory Network*, or treeLSTM. Similar to a GCN, it models graph nodes in terms of their input vectors and the tree structure. While a GCN models the local neighborhood, a treeLSTM models the entire tree. The root node of an encoded tree contains information about the entire tree, and leaf nodes contain information about their path to the root node. We will refer to the treeLSTM syntax encoder as θ_{treeLSTM}.

By stacking enough GCN layers to capture the depth of the entire tree, we can achieve a similar effect. However, up to five stacked GCN layers are needed to fully represent most dependency graphs in the ACE data.[4] It is more natural to treat GCNs as local syntax encoders, with a few layers, and compare them to treeLSTMs, which are inherently designed to encode long-range dependencies.

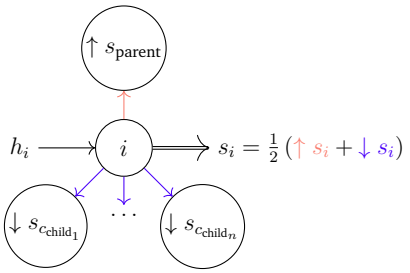
The treeLSTM we use is an extension of the *child-sum tree LSTM* (Tai et al., 2015). In contrast to methods which only encode n-ary trees, it can encode trees with a varying number of children.[5] It was specifically designed to encode syntactic trees. However, we need to extend it in two directions. First, we need to account for dependency labels. In its standard formulation, the child-sum treeLSTM only encodes tree structure, but discards edge labels. Second, we need to define the node representation in terms of children *and* parents. Tai et al. (2015) only look at tasks which require a bottom-up syntax encoding (e.g., sentiment classification in the Stanford Sentiment Treebank, [Socher et al. 2013]), but this is clearly insufficient for event extraction, since many argument candidates are leaves and have no children.

We solve the first problem by incorporating dependency embeddings into the formalism (Section 5.5.2.3). The second problem requires a bi-directional tree encoding. Similar to bi-directional chain-like LSTM, where each node contains information about the entire sequence, a bi-directionally encoded treeLSTM node contains information about all its parents and children.

Before we give the formal definition of our treeLSTM, we want to introduce it in a simpler, ungated form for better apprehension. Consider the following tree.

[4] As described in Section 4.4.1, 888 out of 916 (97%) dependency paths connecting triggers and arguments are of length 5 or less in the ACE test set.

[5] N-ary tree encoding, when applicable, has an advantage over general-tree encoding: Each child position can have its own weights. For example, in binary tree encoding, one can have a left-child and a right-child weight matrix, effectively distinguishing and learning from both. However, dependency graphs cannot be transduced to n-ary trees.

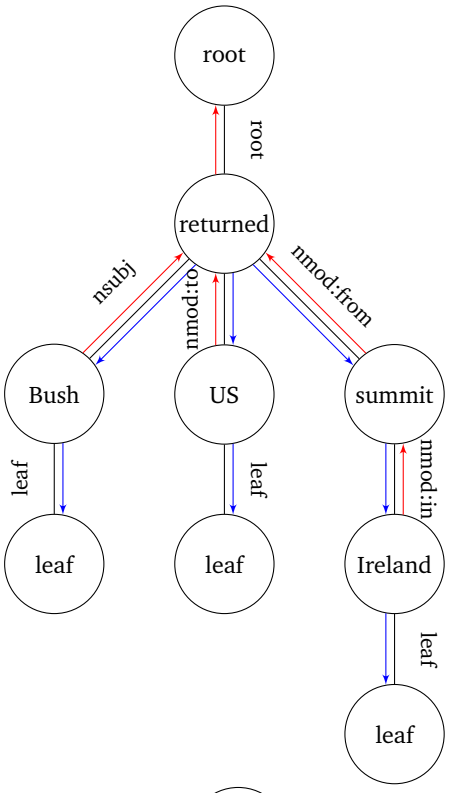


Here, we consider the bi-directional treeLSTM syntax encoding s_i of the node i, which is defined as the average of an 'up' encoding $\uparrow s_i$ and a 'down' encoding $\downarrow s_i$ state. Up and down refer to the processing (or composition) direction of treeLSTMs: The former incorporates the parent state s_{parent} and the latter child states $(s_{\text{child}_1}, \ldots, s_{\text{child}_n})$. A state in a standard bi-directional LSTM (Section 4.3.3.3) is defined by previous and following states. Similarly, a state in a bi-directional treeLSTM is defined by the states of its parent and children. Besides these, i is also defined by its input, h_i (the sentence representation).

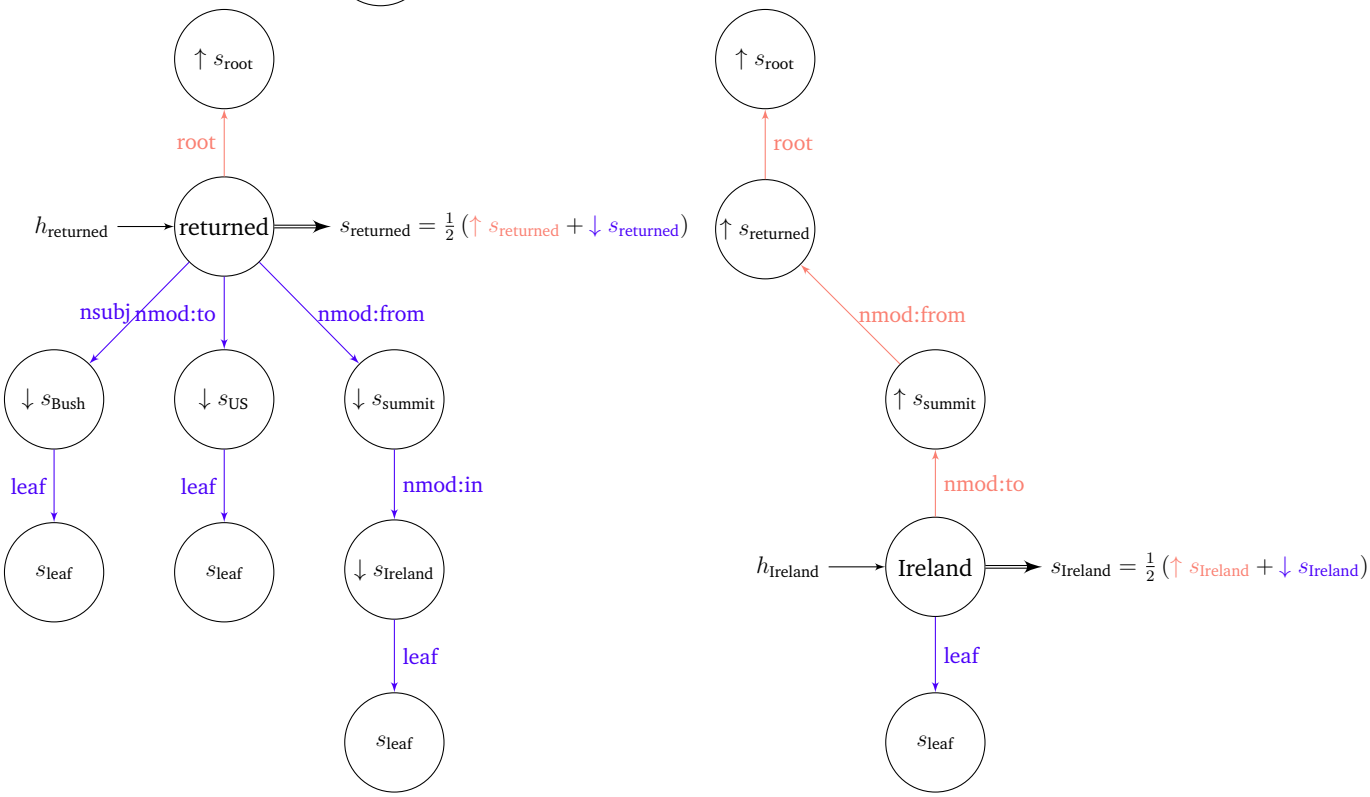
To further illustrate this, consider Figure 5.4. The upper part (5.4a) visualizes the processing flow of a treeLSTM, the lower part (5.4b) illustrate processing flow when computing two syntax-encoded states.

In Figure 5.4a, encoding starts with the only child of the root node ("returned") and its input vector (h_{returned}). The up-treeLSTM processes the parent state of the root node (s_{root}), which is a learnable parameter for the up-treeLSTM. The down-treeLSTM encodes child states, that is, s_{Bush}, s_{us}, and s_{summit}. To produce them, the down-treeLSTM descends to the children of "Bush", "US", and "summit". The children of the former two nodes are leaf nodes, and their state (s_{leaf}) is a learnable parameter. To produce the state for the last node ("summit"), the down-treeLSTM must descend once more to "Ireland", which is again a leaf node. After the down-treeLSTM descends to all leafs, it produces representations for them and uses these representations to build the representations of their parents, etc. The down-representation of "returned" contains information about the entire tree.

Figure 5.4b exemplifies the computation of two states: On the left, we want to produce a syntax encoding for "returned" – we have the input vector h_{returned}, which is produced by the sentence encoder, and the dependency tree. The final encoding state is the average of the respective up- and down-states, produced by the up- and down-treeLSTM. The up-version follows parents until it reaches the root, the down-version follows children until it reaches the leaves. The information flow is depicted by arrows – red represents parent information, blue child information. The encoding



(a) Visualization of treeLSTM processing flow. The dependency tree is depicted in black. Blue edges represent down (↓), red edges up (↑) processing flow. s_{root} and s_{leaf} are learnable parameters; they ensure that every node has a parent and at least one child.



(b) Visualization of producing s_{returned} and s_{Ireland}.

Figure 5.4: Visualization of treeLSTM processing flow (upper part) and information flow when producing two syntax-encoded states (lower part).

state s_{returned} contains the information that its only parent is the root node, and about its subject and nominal modifiers. This information (along with the lexical context, the word vector, and the entity embedding encoded in h_{returned}) help the system to predict that "returned" is a TRANSPORT trigger.

s_{Ireland} (illustrated on the right side of Figure 5.4b) is produced in same way as s_{returned}, using information from parent and child nodes. However, the state encodes less information because "Ireland" is a leaf in the dependency tree. The encoding state contains information about the dependency path to the root, the input vector, h_{Ireland}, and that its only child is the leaf node.

We will now formalize treeLSTMs. We will proceed in the same way we introduced LSTMs in Chapter 4: We first give the definition of a syntax-encoded state $*s_i$ produced by a treeRNN (basically a treeLSTM without gates) and define the respective state produced by treeLSTMs afterwards.

$$* s_i = \sigma \left(*W h_i + *U \sum_{n \in *\mathcal{N}(i)} (*s_n + d_{i,n}) + *b \right). \tag{5.9}$$

In Equation 5.9, σ is a non-linearity, $*W$, $*U$, and $*b$ are weight matrices and a bias, respectively, $*\mathcal{N}(i)$ reports the 'neighbors' of node i, and $d_{i,n}$ gives the embedding vector of the dependency between nodes i and n. Since we use two RNNs to produce the final state s_i (up and down), $*$ is either replaced by $\uparrow$ or $\downarrow$. Note that $\uparrow \mathcal{N}$ returns only parents, and $\downarrow \mathcal{N}$ only children. We combine a dependency embedding and the parent or child state by addition. Conceptually, this is similar to the dependency-parametrized bias of θ_{GCN}. We also tried other composition operators (concatenation, multiplication, abstraction by a one linear layer), but addition gave the best results in all initial experiments. The final syntax-encoded state s_i is the average of the up and down states:

$$s_i = \frac{1}{2} (\uparrow s_i + \downarrow s_i). \tag{5.10}$$

We now give the full definition of our treeLSTM. The equations below will strongly resemble Equation 5.9 above. Note that we cannot use the treeRNN model instead of the treeLSTM because of the same reasons we cannot use RNNs instead of LSTMs: vanishing and exploding gradients (Bengio et al., 1993, Section 4.3.3.2).

Formally,

$$\tilde{y_i} = \sum_{n \in *\mathcal{N}(i)} *s_n + d_{i,n}, \tag{5.11}$$

$$p_i = \sigma(*W_p x_i + *U_i \tilde{y} i + *b_p), \tag{5.12}$$

$$o_i = \sigma(*W_o x_i + *U_o \tilde{y} i + *b_o), \tag{5.13}$$

$$u_i = \sigma(*W_u x_i + *U_u \tilde{y} i + *b_u), \tag{5.14}$$

$$f_{i,k} = \sigma(*W_f x_i + *U_f(*s_k + d_{i,k}) + *b_f), \tag{5.15}$$

$$c_i = p_i \odot u_i + \sum_{n \in \mathcal{N}_i} f_{i,n} \odot c_n, \tag{5.16}$$

$$* \, s_i = o_i \odot \tanh(c_i), \tag{5.17}$$

where in Eq. 4.7, $k \in *\mathcal{N}(i)$. $\odot$ is the Hadamard product. $*$ is again replaced by $\uparrow$ or $\downarrow$, depending on whether it is an up- or a down-treeLSTM. $\uparrow \mathcal{N}(i)$ returns the parents of i, $\downarrow \mathcal{N}(i)$ its children.

As with standard LSTMs, the final state $*s_i$ of i depends on the interaction of the output gate o (Eq. 4.8) with the memory state c, which is itself dependent on the interaction of the other four gates, namely input p (Eq. 4.5), update u (Eq. 4.6) and forget f (Eq. 4.7). $\tilde{y_i}$ accumulates hidden states of parents or children and the respective dependency embeddings. The final system computes two hidden states per node, one up and one down state.

5.6 Training

EVENTOR is sensitive to hyperparameters. It has a narrow range of learning rates for example. We quickly run into exploding gradients even with the treeLSTM if the learning rate is too high. We use a learning rate of 0.0008 (determined by grid search), no weight decay and no regularization.[6] However, regularization is used in the form of online learning and parameter averaging (Section 4.3.8). Again, parameter averaging proves useful to introduce more stability into the training process. Online learning (updating weights after each sample) is necessary because the trees we process with the treeLSTM rarely coincide in shape and therefore are not batchable. We could use

[6] Regularization in the form of $L2$ or dropout did not improve accuracy (Section 5.6.1).

batching with θ_{GCN} and would probably get better performance, but we decided to train both syntax encoders online to increase comparability.

Online training (batch size of 1) is an uncommon training paradigm for deep learning methods. Batch size has a direct effect on training. One would assume that higher batch size increases performance, because parameter updates depend on higher proportions of the training data and therefore lead to a better approximation of the 'ideal classifier' for this data. However, the best performance is sometimes achieved with smaller batch sizes (Keskar et al., 2017; Goyal et al., 2017). Smaller batches introduce more randomness, effectively increasing model generalization. If the batch size is too small however, the positive effect is lost. Smith and Le (2018) show that the scale of introduced randomness is proportional to the batch size, and that there is an optimal batch size, which is much lower than the training set size, but greater than 1. Since we use online learning, our batch size is 1 – this value is probably too low to achieve the best possible performance. We leave it to future research to investigate the effect of online vs. batched learning on our system. In Section 5.6.3 we explain that established methods to decrease overfitting ($L2$ regularization and dropout) do not help our system. We suspect that the reason is online learning. The small batch size masks the effects of $L2$ regularization and dropout.

In the following, we first define our loss function before we report how we deal with class imbalance for trigger types. Finally, we introduce and formalize bagging (Breiman, 1996) – an efficient method to increase performance of complex models when training data is scarce.

5.6.1 Loss

Our loss is the joint negative log-likelihood of all trigger and argument decisions. Specifically, we minimize

$$\mathcal{L} = -\sum_{i=1}^{n} \log p_i - \sum_{i=1}^{n} \mathbb{1}\left(p_i \neq \emptyset \vee p_i^* \neq \emptyset\right) \sum_{j=1}^{k} \log p_{a_{i,j}}, \tag{5.18}$$

for each training instance (event structure within one sentence), where n is the sentence length, k is the entity mention count, p_i is the probability of the correct trigger label at position i, p_i^* is the maximum softmax probability at position i, $p_{a_{i,j}}$ is the correct argument label of entity mention j with respect to trigger label i, and $\mathbb{1}$ is the indicator function. The 'null' label is assignable to both, triggers and arguments.

Our loss function is similar to the loss used by Nguyen et al. (2016), with one (major) difference: They only consider argument losses if the trigger label at position i is not $\emptyset$. This effectively ignores all argument decisions if they rely on false positive triggers and this in turn hinders a system to cope with false positive trigger decisions when predicting arguments. We found a clear preference for our loss in our initial experiments.

The loss described in Equation 5.18 makes an assumption about the learning problem which is violated for event extraction: statistical independence. As Liao and Grishman (2010) show, neither event nor argument type occurrences are independent. The presence of ATTACK for example makes the presence of DIE more likely. Assuming independence keeps the loss function simple however; this is a simplification often made in NLP, and usually performs reasonably well.

Modeling interdependencies is beneficial for event extraction. In fact, this is an important building block of all systems which perform global inference (Li et al., 2013; Yang and Mitchell, 2016; and our work in Chapter 3). One way to model interdependencies, at least within sentences, is to replace the (locally normalized) joint negative log-likelihood by a (globally normalized) *conditional random field* loss (Andor et al., 2016). We leave this interesting aspect to future research.

5.6.2 Repeated Negative Undersampling

Here, we propose a new random undersampling strategy to overcome class imbalance problems. We apply this method to trigger detection training only.

One problem in training neural networks for event extraction is the high class imbalance for triggers. From the 252.212 words in the ACE train set, only 1.3% are triggers (3322), meaning that 98.7% of the words are negative examples[7]. If we break this down to individual event types, the class imbalance becomes even more severe.

There are different possibilities to address this imbalance: Li et al. (2013) ignore sentences which do not contain any events.[8] This is an instance of class undersampling, where the undersampling is not due to chance, but due to some heuristic. This method introduces a high bias towards event types, at least for our system. In our initial experiments, the strategy lowered performance considerably due to an increase in false positive triggers.

[7] The actual number is a bit lower because not each word is allowed as a trigger candidate, e.g,, words within entity mentions cannot be triggers.

[8] This is not reported in their paper, but part of the code we received from them.

Another possibility to overcome class imbalance is to scale the loss: giving non-null classes a higher weight in the final loss drives the model to put more emphasis on these classes. Unfortunately, this resulted in a similar increase in false positives as the first method. Note that loss scaling is equivalent to cost-sensitive learning (Elkan, 2001). There are also more elaborate methods to lower class imbalance. SMOTE (Chawla et al., 2002) for example adds new *minority* class examples by randomly interpolating between pairs of closest neighbors. Liu et al. (2009) present and discuss a few over- and undersampling methods.

We experiment with an easier and presumably more robust method which we call *repeated negative undersampling*. Similar to standard undersampling, we present only a small subset of the majority class (in our case the negative, or non-event class) during training. In contrast to standard undersampling and to the method used in Li et al. (2013), we do not produce *one* undersampled training set. Instead, we repeatedly undersample negative instances throughout training.

Initially, we set an undersampling probability p_n. For each non-trigger word in the *training set*, we draw a number $n \in \mathbb{R}, 0 < n < 1$ from a uniform distribution. If $n < p_n$, we skip the word from training. In the next epoch however, when a new n is drawn for the same word, it might not be lower than the threshold, meaning that this time we would use the negative sample for training.

This method is easy to implement, considerably reduces training time (in contrast to oversampling methods like SMOTE), directly decreases negative effects of class imbalance while still exposing the learner to most negative samples in the training set. This has a positive effect on performance, because it introduces a bias towards non-null cases, but exposes the model to a higher and more diverse amount of null cases compared to exclusive procedures like standard undersampling. We can show in Section 5.7 that repeated negative undersampling has a positive effect on trigger recall, and this in turn positively affects event argument performance. We can measure positive effects across all data splits and syntax encoders.

Initially, we also experimented with repeated negative undersampling for event argument prediction. However, none of our experiments improved performance. The class imbalance problem is less severe for event arguments, because there are more entity mentions which are arguments compared to words which are event triggers. If we undersample the negative arguments class and bias the model towards predicting non-null roles, we introduce many false positive arguments, which in turn greatly weakens argument classification precision.

5.6.3 Bagging

One of the biggest problems with the ACE 2005 data is sparsity – from the 599 annotated documents, 30 are used for development and 40 for testing. There are only 529 documents reserved for training. Data sparsity results in two problems. Limited training data leads to increased model variance and overfitting, resulting in models which do not generalize well because they model spurious characteristics of the training data. Limited test data leads also to higher confidence intervals of evaluation measures, limiting their meaning. Here, we describe how we tackle the first problem. The second problem is addressed by the two new data splits we additionally use for evaluation (Sections 5.7.1 and 5.7).

A common approach to combat variance is to use regularization (James et al., 2013) like $L2$ or dropout (Hinton et al., 2012; Srivastava et al., 2014): Some sort of constraint is applied to the weights during learning, preventing them from growing or shrinking unbounded. $L2$ regularization for example adds the sum of the squared weights to the objective function, with the effect that small weights (close to zero) are preferred over (positively or negatively) large weights. This typically increases generalization. In our initial experiments, regularization in the form of $L2$ and dropout did not increase performance. Either the regularization strength was too low to show effects, or it quickly became too high and deteriorated performance. We suspect that our system needs parameters in a narrow value range not centered around 0 to perform well. Higher $L2$ regularization forces values to be small and fall out of that range. Due to online learning, we cannot use common methods like batch normalization (Ioffe and Szegedy, 2015) to combat this behavior.

Another method to combat variance is to use ensembles. All ensemble methods have in common that they train and use multiple kinds of classifiers. *Boostrap aggregating*, or *bagging* (Breiman, 1996) in particular was designed to minimize variance in the context of limited training data using only one type of classifier trained on multiple versions of the training data, sampled randomly.

High variance goes along with 'instability'. In Breiman (1996), a stable training procedure is one which converges to similar solutions when the training set is altered slightly. Consequently, an unstable procedure reaches to very different solutions for small changes. Training neural networks is an indeterministic process, and the models usually consist of a large number of parameters. We expect variance to be high if we train such models on a limited amount of training data. To test if our training is

stable, we train five models on the training set. We then measure *cosine similarity* of the resulting pre-softmax trigger and argument layers of our system.[9] To be more specific, we concatenate the upmost weight vectors for trigger and argument classification (without syntax encoders) for the five models we train and compute their mean cosine similarity. Each vector has 37800 elements, each a trainable parameter. We measure an average similarity of 0.43 between the five vectors. To see if this number itself is stable, we train another 25 models and measure an average similarity of 0.42. Note that the parameter averaging we use (Section 4.3.8) smooths noise out and lowers variance. Training yet another 25 models *without* parameter averaging results in an average cosine similarity of 0.30. A stable training would produce a similarity close to 1. From this, we can conclude that our training is rather unstable, and small variances to the training set result in different weight vectors high up in the neural architecture. With bagging, we can leverage this fact and train dissimilar models which vote for the final class, effectively improving over single models.

Bagging itself is simple: Combine and randomize the training and development sets, split it into a new training and development sets, and select the best model trained on the new split. This is repeated k times, resulting in k classifiers. Note that one document may be (by chance) part of up to k train sets, or excluded from all. The ensemble of k models uses majority voting to come to a final conclusion when predicting. This ensemble is the final form of our system. We will now formalize bagging.

[9] Cosine similarity is defined as

$$\text{sim}(\vec{u}, \vec{v}) = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|_2 \, \|\vec{v}\|_2},$$

where $\vec{u}$ and $\vec{v}$ are two vectors (in our case, the concatenated weights of the two highest layers in our system) and $\|\bullet\|_2$ is the 2-norm of the vector. Cosine similarity ranges from -1 to 1, with 0 meaning 'no similarity'.

Algorithm 5: BAGGING (training set $\mathcal{T}$, folds k, split ratio r, training epochs e)

ensemble $\leftarrow \emptyset$;
for $1 \dots k$ **do**
  $\hat{\mathcal{T}} \leftarrow$ RANDOMIZE($\mathcal{T}$);
  /* The split ratio determines the size of τ_{dev} compared to τ_{train} */
  $\tau_{\text{train}}, \tau_{\text{dev}} =$ SPLIT($\hat{\mathcal{T}}, r$);
  /* TRAIN returns the best model on τ_{dev} */
  model $=$ TRAIN($\tau_{\text{train}}, \tau_{\text{dev}}, e$) ;
  ensemble $\cup$ model
return ensemble

Input to Algorithm 5 is a training set $\mathcal{T}$, the number of train set subsamples k (which is also the number of classifiers in the ensemble), the ratio r between (subsampled) training and development documents, and the number of training epochs e. Note that $\mathcal{T}$ contains all documents which are not part of the widely used test set; bagging enables us to have 30 more documents for training compared to the standard procedure.

5.7 Experiments and Results

Please refer to Section 2.4 for criteria when trigger and argument decisions are correct. We again adopt the proposal we make in Section 2.5.2 here: We always train and evaluate five times, and report evaluation numbers averaged over these five runs. Furthermore, we report sample standard deviations.

In this section, we want to analyze different aspects of EVENTOR's performance. First, we show the effects of parameter averaging and obtain an optimal value for repeated negative undersampling on the development set of the widely used data split. Then, we come to the main experiment of this chapter, the comparison of the different syntax encoders ($\theta_{\emptyset}$, θ_{GCN}, θ_{treeLSTM}). Throughout this section, we follow the same evaluation standards we used in Chapter 4, introduced and discussed in Section 2.5. However, to make the numbers obtained in our main experiment more reliable, we additionally construct two new data splits and evaluate the syntax encoders on all three, the newly constructed as well as the well established splits. The problem with the latter is that its test set ignores most genres in ACE and only consists of newswire

articles. Section 5.7.1 describes how we construct the new splits and reports genre distribution statistics. Then, we evaluate the best syntax encoders against the current state-of-the-art. Finally, we analyze the effects of bagging.

In particular, we devise the following experiments.

- **Experiment 1**: Test effects of parameter averaging and repeated negative undersampling on the development set of Split 1 for the base system ($\theta_{\emptyset}$)
- **Experiment 2**: Evaluate trigger and argument classification performances of all encoders ($\theta_{\emptyset}$, θ_{GCN}, θ_{treeLSTM}) on the test sets of all three splits
- **Experiment 3**: Compare θ_{GCN} and θ_{treeLSTM} against the state-of-the-art on the test set of Split 1
- **Experiment 4**: Compare the effects of bagging for the base system ($\theta_{\emptyset}$) on the test set of split 1

Experiment 1 has the purpose to investigate the effects of parameter averaging and repeated negative undersampling on the base system, i.e., EVENTOR without a syntax encoder ($\theta_{\emptyset}$). We find that both factors have a great impact on performance. The effects of parameter averaging are not surprising because no other regularization method worked in our online training setting. We were surprised however to find that our undersampling approach has a profound impact on results. As we will discuss, the effect is linked to an increase in trigger recall, which in turn positively affects argument performance – something we already observed in Chapter 3.

Experiment 2 is the main experiment in this chapter. It answers the question if syntax encoders can help event extraction as a whole. We already showed that such encodings can benefit event argument classification (Chapter 4), at least under 'laboratory conditions' with given event triggers. We find a positive effect, which is, however, smaller than in chapter 4 in terms of absolute numbers. Because our undersampling method has such a profound impact on results, we do not only report numbers for the optimal undersampling probability we obtained in Experiment 1, but also for the other tested probabilities.

Experiment 3 is a follow-up to Experiment 2 and tests the average and best evaluation runs of θ_{GCN} and θ_{treeLSTM} against five other event extractors, including the base system from Chapter 3. We find that both encoders perform *on average* on a par with the best event extractors published to date. One problem with this comparison is that

		Split 1			Split 2			Split 3		
	total	train	dev	test	train	dev	test	train	dev	test
documents	599	529	30	40	399	100	100	399	100	100
broadcast news	0.38	0.41	0.27	0.00	0.37	0.47	0.30	0.39	0.33	0.39
weblogs	0.20	0.22	0.17	0.00	0.20	0.20	0.20	0.20	0.22	0.18
newswire	0.18	0.11	0.27	1.00	0.17	0.17	0.20	0.16	0.22	0.20
broadcast conversation	0.10	0.10	0.17	0.00	0.11	0.05	0.12	0.09	0.10	0.13
usenet newsgroups	0.08	0.09	0.12	0.00	0.08	0.05	0.12	0.09	0.07	0.04
telephone speech	0.06	0.07	0.0	0.00	0.07	0.06	0.06	0.07	0.06	0.06

Table 5.1: Split statistics for the widely used data split ('Split 1') and two additional splits which follow the distribution of ACE genres more closely. The first line reports the numbers of document for each train/dev/test set, the other lines report ratios of documents belonging to the respective genre (ratios for one column add up to 1).

we do not have information about the exact nature of the experiments carried out in the other publications. They do not report standard deviations and do not mention multiple training and testing runs. This leaves us with two assumptions: They either reported the first and only evaluation run, or they reported the best out of a few runs. Therefore, we also report the best evaluation run of both encoders.

Experiment 4 finally is a side experiment which shows the effects of bagging on the standard test set. This is hard to evaluate on the development set because bagging re-splits data n times, resulting in n different development sets. However, the test set remains unchanged.

Before we discuss results, we want to specify why and how the additional data splits we use for Experiment 2 are constructed.

5.7.1 Resampling New Splits

The standard test set introduced by Ji and Grishman (2008) and discussed in Section 2.2.6 ignores five of the six genres in ACE. It consists only of newswire articles and neglects weblogs, broadcast news, broadcast conversations, Usenet newsgroups, and telephone conversation transcripts. Furthermore, it is quite small: 40 documents (6.7% of ACE) consisting of 440 triggers and 916 arguments. We believe that this test set alone is not enough to represent the ACE data. We decided to produce two additional data splits, such that each new train/development/test set closely follows the overall genre distribution. The procedure is formalized in Algorithm 6.

Algorithm 6: SPLIT DATA (ACE 2005 data A, dev set size n_{dev}, test set size s_{test})

```
1  mle←MAXIMUM LIKELIHOOD GENRE ESTIMATES(A);
   /* initially, every document is in the train set                    */
2  train← A;
3  dev← ∅;
4  test← ∅;
5  while |dev| < n_dev do
       /* draw random document                                         */
6      a ← RANDOM(A);
       /* get the genre probability (maximum likelihood estimate)      */
7      p_genre ← mle(GENRE(a));
       /* {r ∈ ℝ|0 < x < 1}                                            */
8      if random number r < p then
           /* add a with a certain probability                         */
9          dev=dev∪a;
10         train=train∧a;
11 while |test| < n_test do
12     ...
13 return train, dev, test
```

Table 5.1 reports statistics about the original, widely-used split ('Split 1'), as well as the two new splits ('Split 2' and 'Split 3'). There are two main points encoded in Table 5.1: First, the new splits use more than twice as many documents for development and testing than the original split. Second, the new train/dev/test sets follow the overall genre distribution, whereas the original sets do not – especially the original test set only contains newswire articles. In the following, make extensive use of the new splits and devise a thorough evaluation of syntax encoding methods for event extraction.

Randomly re-sampling data splits is akin to bootstrapping for validation (Efron, 1992) with a non-uniform, genre-based maximum likelihood sampling probability. We want our new splits to closely follow the overall ACE genre distribution in order to evaluate method and model behavior more accurately compared to the standard split, where the test set ignores most document genres. This approach is similar to stratified

	Trigger Classification						Argument Classification					
p_n	0.0		0.5		0.9		0.0		0.5		0.9	
prm avg	-	+	-	+	-	+	-	+	-	+	-	+
1	62.6	68.5	59.2	69.5	66.2	70.6	46.1	53.7	44.1	53.3	46.3	56.3
2	63.3	68.7	56.0	68.3	64.9	70.4	46.9	54.2	43.9	53.0	47.3	56.5
3	63.4	69.2	59.9	68.2	64.4	69.8	46.7	52.8	45.7	52.4	47.0	55.6
4	63.6	69.8	61.0	69.1	63.0	70.8	45.2	54.6	44.4	53.7	44.7	55.9
5	64.1	69.1	60.2	69.0	67.0	70.6	47.0	55.2	42.1	53.9	50.0	55.3
$\bar{x}$	63.4	69.1	59.3	68.8	65.1	70.4	46.4	54.1	44.0	53.3	47.1	55.9
σ	0.5	0.5	1.9	0.6	1.6	0.4	0.7	0.9	1.3	0.6	1.9	0.5

Table 5.2: **Experiment 1.** Development set F_1 for trigger and argument classification on the standard split (Split 1) for $\theta_\emptyset$. "p_n" reports the repeated negative undersampling probability. "prm avg" specifies whether parameter averaging is used. "$\bar{x}$" reports the average of the five training/testing runs and "σ" the respective sample standard deviation.

cross validation.[10] We decided to produce two new random splits because this significantly reduced the amount of training and testing time compared to, e.g., 10-fold cross validation. Furthermore, we are not interested in aggregated evaluation results over multiple folds but in the relative performance differences of different syntax encoders for event extraction.

5.7.2 Experiment 1

Experiment 1 tests the effects of parameter averaging (Section 4.3.8) and repeated negative undersampling (Section 5.6.2) on the development set of Split 1. We evaluate EVENTOR with no syntax encoder ($\theta_\emptyset$) and no bagging.

Table 5.2 reports trigger and argument classification F_1 scores on the development set of the standard Split 1 for three repeated negative undersampling probabilities ($0.0, 0.5, 0.9$, indicated in Line 1) without and with parameter averaging (indicated by "-" and "+" in Line 2). We report the F_1 scores of all five training and testing rounds, as well as an average score ("$\bar{x}$") and the sample standard deviation ("σ").

The first thing we notice is that the difference without and with parameter averaging is very large. For trigger classification, the improvement with parameter averaging is about 6, for argument classification it is around 8 F_1 points. Please note that we optimized the learning rate for the system *with* parameter averaging, meaning that

[10] However, in contrast to cross validation, we sample with replacement.

in the optimal case the absolute difference is probably smaller. However, we consistently measure a significantly better performance when using parameter averaging. All further experiments are carried out using it.

Repeated negative undersampling also has a positive effect on performance. While we notice a performance drop for undersampling probability $p_n = 0.5$, we notice an increase for both, trigger and argument predictions for $p_n = 0.9$. Therefore, we choose $p_n = 0.9$ as the undersampling probability for all other experiments. In Experiment 2 however, where we test the contributions of syntax encoding in general, we also report evaluation results for other p_n values to get a more complete view of the contributions of our undersampling method. We also discuss its impact on event extraction in more detail there. We can note however that we get an increase in about 1.5 F_1 points for triggers and arguments on the development set when using $p_n = 0.9$.

5.7.3 Experiment 2

Here, we evaluate trigger and argument classification performance of our syntax encoders, including no encoding. The purpose of this experiment is to investigate the contribution of syntax encodings to event extraction. We evaluate their contribution on the test sets of all three data splits. Furthermore, we use parameter averaging, bagging, and set $p_n = 0.9$. We also report results for the other undersampling probabilities to get a more complete picture of the impact of negative undersampling on event extraction. The setting with $p_n = 0.9$ is our main setting and the only one we use in Experiment 3, where we compare our syntax encoders to the state-of-the-art.

Tables 5.3 and 5.4 report trigger and argument classification performances across the three data splits, the three syntax encoders, and three negative undersamplig probabilities. In order to make the vast amount of numbers more easily understandable, we visualize them in Figure 5.5 for trigger classification and in 5.6 for argument classification. Numbers for $p_n = 0.9$, our actual evaluation setting, are printed in bold. As mentioned above, we also report numbers for $p_n = 0.0$ and $p_n = 0.5$ to get a more complete picture of the impact of negative undersampling.

The main question we want to answer with Experiment 2 is if and how much our syntax encoders contribute to improving event extraction performance. Figures 5.5 and 5.6 show trigger and argument classification precision, recall, and F_1 for each syntax encoder (blue: $\theta_\emptyset$, red: θ_{GCN}, green: θ_{treeLSTM}) on all data splits. The points at 0.9 represent the main evaluation points because we determined that $p_n = 0.9$ is the

		$\theta_\emptyset$			θ_{GCN}			θ_{treeLSTM}		
	p_n	0.0	0.5	**0.9**	0.0	0.5	**0.9**	0.0	0.5	**0.9**
Split 1	P	77.5	77.0	**68.7**	78.1	76.1	**68.5**	78.5	76.9	**69.2**
	R	63.1	65.2	**68.8**	63.7	64.6	**69.9**	61.6	64.2	**69.6**
	F_1	69.5	70.6	**68.7**	70.2 ‡	69.9	**69.2** †	69.0	69.9	**69.4**
Split 2	P	71.7	72.2	**63.2**	70.9	70.7	**63.0**	71.9	72.2	**65.1**
	R	62.3	63.8	**71.0**	63.7	65.1	**71.2**	62.2	63.0	**69.9**
	F_1	66.6	67.8	**66.9**	67.1 †	67.8	**66.8**	66.7	67.3	**67.4** ‡
Split 3	P	74.4	74.4	**67.1**	73.8	73.8	**67.0**	73.3	73.1	**67.0**
	R	63.5	64.1	**72.0**	63.4	65.5	**72.4**	64.6	66.0	**72.4**
	F_1	68.5	68.9	**69.5**	68.3	69.4 †	**69.6**	68.7 ‡	69.4 †	**69.6**

Table 5.3: **Experiment 2.** Trigger classification precision (P), recall (R), and F_1 for our three syntax encoders ($\theta_\emptyset$, θ_{GCN}, θ_{treeLSTM}) and three negative undersampling probabilities $(0.0,\ 0.5,\ 0.9)$. The columns where $p_n = 0.9$ are our main evaluation numbers and printed in bold. Each evaluation number is the average of five independent training and testing rounds. † means better F_1 score compared to the respective $\theta_\emptyset$ score, statistically significant with $p < 0.05$ (‡ : $p < 0.1$). For significance, we average evaluation numbers across the five models.

		$\theta_\emptyset$			θ_{GCN}			θ_{treeLSTM}		
	p_n	0.0	0.5	**0.9**	0.0	0.5	**0.9**	0.0	0.5	**0.9**
Split 1	P	59.4	61.2	**58.0**	58.9	59.4	**57.6**	58.5	59.5	**57.4**
	R	48.0	50.3	**51.5**	50.8	51.0	**54.2**	48.8	50.9	**52.8**
	F_1	53.0	55.2	**54.5**	54.5†	54.9	**55.8**†	53.2	54.9	**55.0**
Split 2	P	57.9	59.3	**54.4**	57.7	57.4	**53.8**	56.7	58.3	**55.5**
	R	48.0	47.9	**54.0**	49.7	50.5	**55.7**	49.5	48.4	**54.4**
	F_1	52.4	53.0	**54.2**	53.4 †	53.7	**54.7**‡	52.8	52.9	**55.0**
Split 3	P	65.5	65.3	**63.4**	64.4	65.3	**62.0**	64.8	64.2	**62.4**
	R	50.6	51.3	**57.3**	52.5	53.4	**59.2**	53.2	54.9	**58.2**
	F_1	57.1	57.5	**60.1**	57.9 ‡	58.7 †	**60.5**	58.4 †	59.2 †	**60.2**

Table 5.4: **Experiment 2.** Argument classification precision (P), recall (R), and F_1 for our three syntax encoders ($\theta_\emptyset$, θ_{GCN}, θ_{treeLSTM}) and three negative undersampling probabilities $(0.0,\ 0.5,\ 0.9)$. The columns where $p_n = 0.9$ are our main evaluation numbers and printed in bold. Each evaluation number is the average of five independent training and testing rounds. † means better F_1 score compared to the respective $\theta_\emptyset$ score, statistically significant with $p < 0.05$ (‡ : $p < 0.1$). For significance, we average evaluation numbers across the five models.

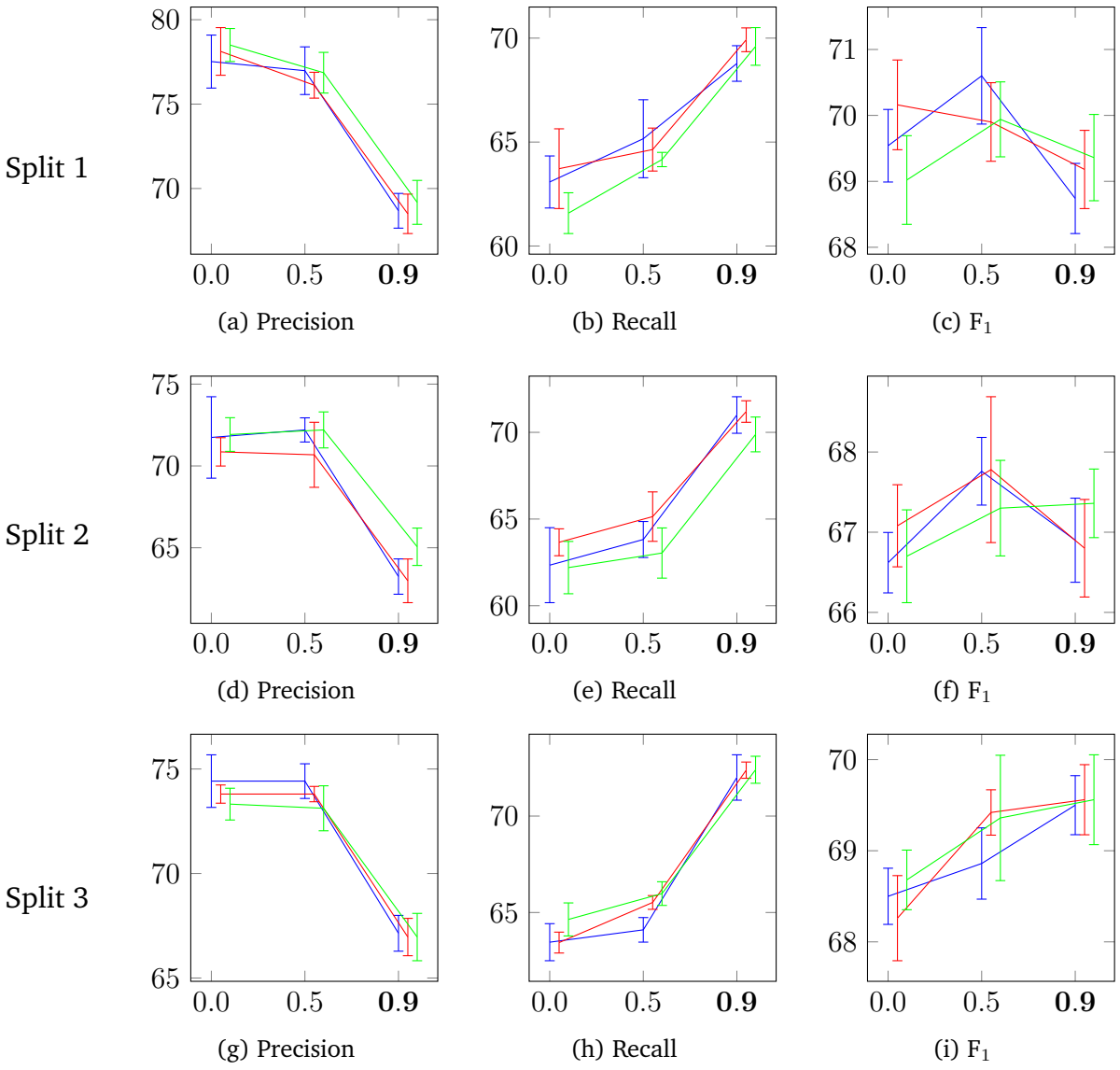


Figure 5.5: **Experiment 2.** Trigger classification performance. Each point reports the average results of five models for $\theta_{\emptyset}$ (blue), θ_{GCN} (red), and θ_{treeLSTM} (green) on the same test set. Y axes report evaluation scores. X axes report negative undersampling probabilities (0.0, 0.5, 0.9). The points at 0.9 represent our main evaluation numbers. Each row reports results on a different data split, with the first row ("Split 1") being the widely adopted split. Error bars are sample standard deviations.

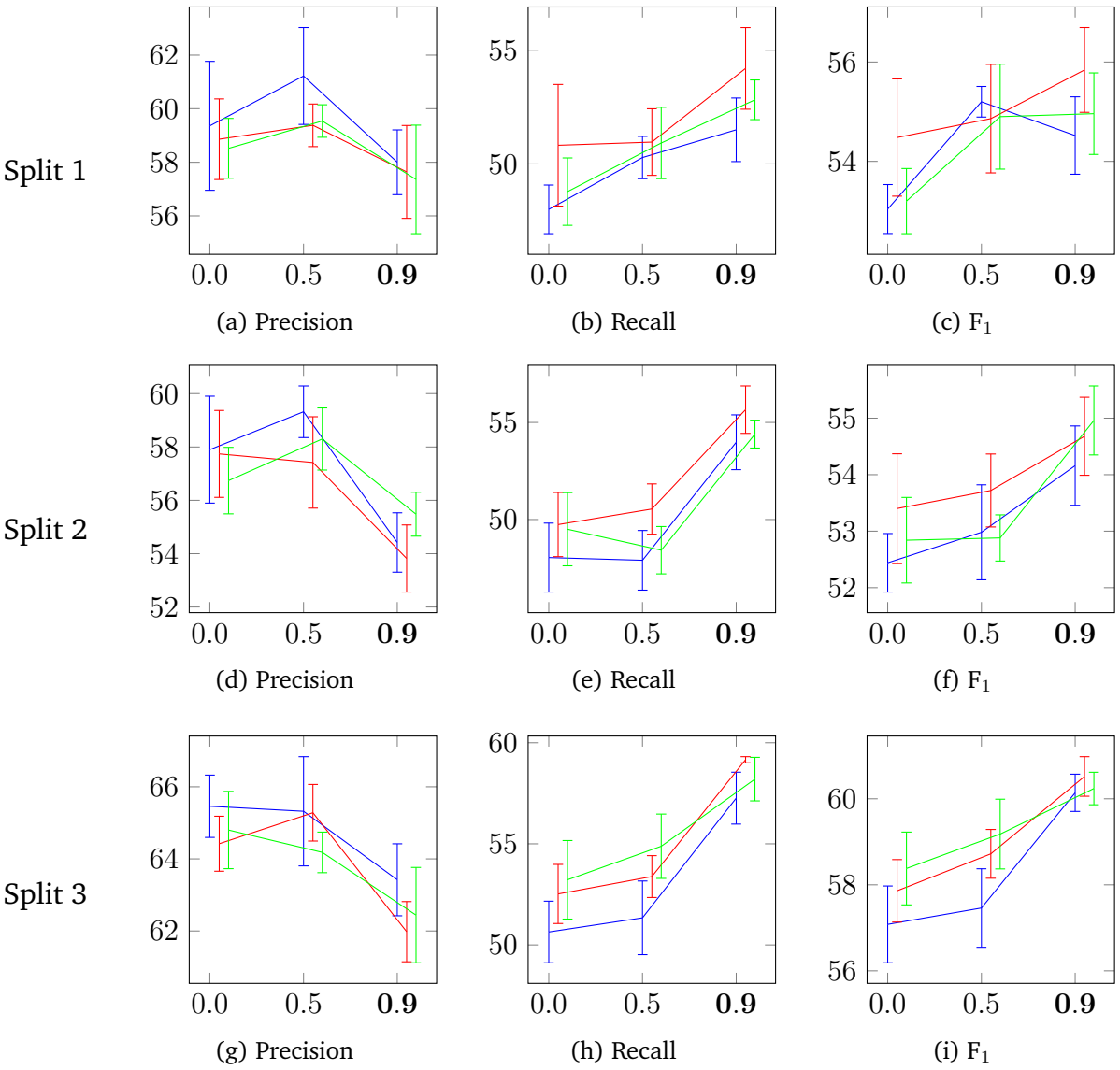


Figure 5.6: **Experiment 2.** Argument classification performance. Each point reports the average results of five models for $\theta_{\emptyset}$ (blue), θ_{GCN} (red), and θ_{treeLSTM} (green) on the same test set. Y axes report evaluation scores. X axes report negative undersampling probabilities (0.0, 0.5, 0.9). The points at 0.9 represent our main evaluation numbers. Each row reports results on a different data split, with the first row ("Split 1") being the widely adopted split. Error bars are sample standard deviations.

		Trigger Classification			Argument Classification		
p_n		0.0	0.5	0.9	0.0	0.5	0.9
Split 1	θ_{GCN}	**0.09**	0.64	**0.02**	**0.00**	0.66	**0.03**
	θ_{treeLSTM}	0.97	0.58	0.53	0.97	0.58	0.53
Split 2	θ_{GCN}	**0.04**	0.11	**0.09**	**0.04**	0.12	**0.09**
	θ_{treeLSTM}	0.47	0.60	**0.07**	0.46	0.58	0.10
Split 3	θ_{GCN}	**0.00**	**0.00**	0.50	**0.09**	**0.01**	0.50
	θ_{treeLSTM}	**0.06**	**0.00**	0.95	**0.00**	**0.00**	0.96

Table 5.5: Statistical significance scores (p-values) for trigger and argument classification. We test the null hypothesis that the differences in F_1 of a syntax encoder (θ_{GCN} or θ_{treeLSTM}) and $\theta_\emptyset$ are due to chance. We always compare a syntax encoder with $\theta_\emptyset$ on the same split and with the same p_n values. We print all values with $p < 0.1$ in bold, and we additionally underline all values with $p < 0.05$. We use approximate randomization (Noreen, 1989).

optimal undersampling probability on the development set. We will discuss results for the other points afterwards. The numbers for triggers and arguments are reported in Tables 5.3 and 5.4, respectively.

We start with the analysis of trigger classification performance (Figure 5.5, Table 5.3). In terms of F_1, θ_{treeLSTM} (green) is either better (for Splits 1 and 2) or on a par with $\theta_\emptyset$ (Split 3). The improvements are +0.7 F_1 for Split 1, +0.5 F_1 for Split 2, and + 0.1 for Split 3. θ_{GCN} is either on a par (Splits 2 and 3) or above $\theta_\emptyset$. We can note that the syntax encoders have a slight positive effect on trigger classification performance, especially θ_{treeLSTM}.

When we look at argument classification performance (Figure 5.6, Table 5.4), we see a similar trend: syntax encoders improve results. However, here, θ_{GCN} works better than θ_{treeLSTM} on Splits 1 and 3. The improvement compared to $\theta_\emptyset$ is +0.4 F_1 for both. For split 2, θ_{treeLSTM} has an increased performance of +0.8 and θ_{GCN} of +0.5 F_1. In general, argument recall is better for syntax encoders, suggesting that syntax representations enable the event extractor to find more arguments. This also includes the observation that syntax encoders lead to more false positives,which is reflected in a lower argument precision compared to $\theta_\emptyset$.

Table 5.5 reports statistical significance (p-values, approximate randomization) for trigger and argument classification F_1 scores of θ_{GCN} and θ_{treeLSTM} compared to $\theta_\emptyset$. We compare the methods/systems on the same split and p_n value. The p-value is the probability that we observe similarly or more extreme differences if the null hypothesis

(differences in F_1 score is due to chance) is true. The difference in trigger classification F_1 of $\theta_\emptyset$ and θ_{GCN} on Split 1 is 1.5 points for example. We have a chance of about 9% (0.09) that we would observe such a difference by random luck. We print all values with $p < 0.1$ in bold, and additionally underline all values with $p < 0.05$.

Please note that it is not straightforward to compute statistical significance in our case because we report evaluation scores which are averaged across five different training and testing rounds. We decided to average the evaluation scores as well before we execute approximate randomization instead of, e.g., computing significance for each run compared to each $\theta_\emptyset$ run and averaging the scores afterwards, or regard an averaged difference as significant if most of the individual differences are significant. Averaging the scores beforehand makes it more difficult to reach statistical significance because extreme values are averaged out. Therefore, we decided to include $p < 0.1$-values if we speak of significance. Usually, only values with $p < 0.05$ are regarded 'statistically significant' in a strict sense.

Usually, trigger and argument classification F_1 score differences are both either significant or not. For $p_n = 0.9$, θ_{GCN} produces significant results for triggers and arguments on Splits 1 and 2. For $p_n = 0.0$, θ_{GCN} has significant results for all splits. θ_{treeLSTM} on the other hand has only significant results on Split 3 if $p_n < 0.9$, and on Split 2 for trigger classification if $p_n = 0.9$. We can conclude that θ_{GCN} tends to have better and more significant results compared to θ_{treeLSTM}.

The second point we want to investigate in Experiment 2 is a side-question: How severe is the impact of negative undersampling on event extraction? To answer this, we also report results for $p_n = 0.0$ and $p_n = 0.5$. For trigger classification (Figure 5.5), the results are clear: undersampling improves results considerably across splits and methods. However, Splits 1 and 2 show the best performance at 0.5 and not at 0.9, as we determined on the development set. Such deviations between development and test set are to be expected. Nevertheless, argument classification performance (Figure 5.6) is always best at 0.9 across almost all splits and methods. This is counter-intuitive: We expect argument performance to be best where trigger F_1 is best. The reason why this is not the case is trigger recall, a result we also have in Chapter 3 where we improve argument performance because we can improve trigger recall. The argument decoder profits considerably from finding more triggers (or: increasing trigger recall) because every missed trigger automatically leads to missing all of the respective arguments. A few false positive triggers which come into play when trigger recall is increased do not change this effect. As already mentioned, we already observe

this effect in Chapter 3. Here, we want to investigate it further, since it appears again in a very different system.

To measure the dependence of trigger recall and argument F_1, we measured the Pearson correlation between all trigger recall and their respective argument F_1 scores across all splits and p_n values. We measure a correlation of 0.5, which is usually interpreted as a 'moderate positive correlation'. The correlation between trigger precision and argument F_1 is -0.2, a slightly negative correlation.

Please note that correlation does not imply causation. The positive correlation of trigger recall and argument F_1, as well as the inverse relationship between trigger precision and trigger recall on the one hand (trigger precision always drops when recall increases), and the negative correlation between trigger precision and argument F_1 on the other, which hold across all splits and methods, lead us to our second finding: Increasing trigger recall leads to better argument classification performance.

To come back to our undersampling method: Increasing p_n leads to increasing trigger recall, and by extension to increasing argument prediction performance. In general, syntax encoders improve argument classification F_1 scores more for p_n values 0.0 and 0.5, although absolute numbers are best for 0.9. Another observation is that the higher argument classification recall of θ_{treeLSTM} and θ_{GCN} is more pronounced for lower p_n values. This is an effect of the downside of increasing trigger recall by increasing a system's bias towards predicting events – recall rises, but precision decreases, meaning that the system predicts more false positive events along with the increased true positive set, and these false positives lead to false positive arguments. With lower p_n, this effect is also smaller. From this, we can conclude that a method which increases overall performance of trigger classification (precision and recall) would greatly benefit from syntax representations – concluding from our three test sets, on average about 0.6 F_1 points for trigger and more than 1 F_1 point for argument classification.

We can conclude the main question of this experiment and of the entire chapter: Syntax encoders have a positive effect on argument predictions, although not as clear cut as in Chapter 4, where we used gold triggers. We expect that syntax encoder benefits increase with higher trigger performance, especially if the increase is not only in trigger recall, but also in trigger precision. In this evaluation, the effect of syntax encoders at least partially masked by the high impact of negative undersampling.

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
$\theta_\emptyset$ **avg**	71.5	71.6	71.5	68.7	68.8	68.7	60.6	53.8	57.0	58.0	51.5	54.5
θ_{GCN} **avg**	71.7	72.6	71.8	68.5	69.9	69.2 †	59.9	56.3	58.1	57.6	54.2	55.8 †
θ_{treeLSTM} **avg**	71.7	72.1	71.9	69.2	69.6	69.4	59.9	55.2	57.4	57.4	52.8	55.0
$\theta_\emptyset$ **max**	72.0	72.0	72.0	68.9	68.9	68.9	61.8	54.3	57.8	59.3	52.1	55.5
θ_{GCN} **max**	72.4	72.7	72.6	69.7	70.0	69.8 †	60.0	58.8	59.4	57.6	56.6	57.1 †
θ_{treeLSTM} **max**	80.2	65.5	72.1	78.6	64.1	70.6 †	62.5	55.1	58.6	60.5	53.4	56.7 †
Li et al. (2013)*	76.9	65.0	70.4	73.7	62.3	67.5	69.8	47.9	56.8	64.7	44.4	52.7
Chen et al. (2015)⋆	80.4	67.7	73.5	75.6	63.6	69.1	68.8	51.9	59.1	62.2	46.9	53.5
Nguyen et al. (2016)⋆	68.5	75.7	71.9	66.0	73.0	69.3	61.4	64.2	62.8	54.2	56.7	55.4
Zhang et al. (2017)⋆	n/a	n/a	n/a	75.1	64.3	69.3	n/a	n/a	n/a	63.3	50.1	55.9
Chen et al. (2017)⋆‡	79.7	69.6	74.3	75.7	66.0	70.5	71.4	56.9	63.3	62.8	50.1	55.7

Table 5.6: Micro-averaged performance of EVENTOR against most recent systems. "Avg" refers to result averaged over five training/testing runs. "Max" refers to the overall best run. All EVENTOR numbers are obtained with $p_n = 0.9$. * means that the system uses the same feature set as EVENTOR. ⋆ means that the evaluated models are produced by non-deterministic training, but it was not specified in the publication how the evaluation numbers were obtained (average across multiple runs, best, random, etc.). ‡ means that non-ACE data was additionally used for training. † means statistically significant score ($p < 0.05$) compared to $\theta_\emptyset$. We only tested significance for classification F_1 scores.

5.7.4 Experiment 3

Experiment 3 compares our syntax encoders against the state-of-the-art. Table 5.6 reports the results for EVENTOR against the most recent neural event extractors. We also include evaluation results from Li et al. (2013) for reference because EVENTOR uses the same argument feature set as they do.

All reported numbers are obtained with parameter averaging, bagging, and $p_n = 0.9$. "Max" refers to the best EVENTOR run using the respective encoder; "avg" lines report results averaged over five runs. ⋆ means that the respective publication does not specify how results were obtained (averaging across multiple runs, best run, random run, etc.). It is therefore not clear how the evaluation numbers marked with ⋆ compare to our results. All evaluations in Table 5.6 are obtained on the standard split (Split 1).

When we compare θ_{GCN} and θ_{treeLSTM} against $\theta_\emptyset$, we see that the syntax encoders perform better on average as well as in the best case (max). The max and the average

	Trigger Classification			Argument Classification		
	P	R	F_1	P	R	F_1
no bagging	70.8±1.4	66.3±0.7	68.5±0.6	53.9±1.7	51.8±1.1	52.8±1.1
bagging	68.7±1.0	68.8±0.9	68.7±0.5	58.0±1.2	51.5±1.4	54.5±0.8

Table 5.7: **Experiment 4.** Test set trigger and argument classification precision (P), recall (R), and F_1 with and without bagging for $\theta_\emptyset$. Each evaluation number is the average of five independent training and testing rounds.

θ_{GCN} F_1 differences to $\theta_\emptyset$ are statistically significant with $p < 0.05$ using approximate randomization. As previously, we only test classification F_1 significance.

Average θ_{GCN} results are comparable to all other neural event extractors, with maximum argument prediction results well above the current state-of-the-art. Average θ_{treeLSTM} results show a similar trend, even though argument performance is a bit under the current state-of-the-art.

However, as we discuss in Section 2.5, maximum results are *not* a proper representative of a neural system's performance. Further evidence for this claim comes from a recent publication (Liu et al., 2018) which reports average trigger classification results for Chen et al. (2015), a seminal paper for 'neural' event extraction, obtained via five-fold cross validation instead of the original unspecified setting: The original paper reports a 69.1 trigger classification F_1 (see also Line 6 in Table 5.6), while Liu et al. (2018) report 64.9 F_1 score for the same system and five-fold cross validation.

5.7.5 Experiment 4

In our last experiment, we want to show the effects of bagging on $\theta_\emptyset$ on the standard test set (Split 1) with $p_n = 0.9$.

Table 5.7 reports trigger and argument classification results (precision, recall, F_1) with and without bagging on the standard test set (Split 1). As we can see, bagging improves trigger recall (+2.5) and argument precision (+4.1). Please note that these performance improvements are on top of the improvements from parameter averaging. We test the effect of bagging on the test set because the method re-splits training and development set for each fold anew, resulting in n different development sets. Fixing the development set would require that bagging works with less training data, and this in turn might destroy performance differences. The test set however is always fix and does not change because of bagging.

5.8 Conclusion

In contrast to Chapter 4, we investigate the use of syntax representations for the entire event extraction task, including trigger identification and classification. Additionally, we transform the points we discuss in Section 2.5 into evaluation schemes and extend standard evaluation procedures to two new data splits which closely follow the overall ACE genre distribution for each train/dev/test set, reaching a reliable evaluation of our work. We evaluate all methods we compare ($\theta_{\emptyset}$, θ_{GCN}, θ_{treeLSTM}) within one system, especially using the same preprocessing, on multiple data splits, always averaging five runs of five independently trained models before we report an evaluation number.

We find that increasing the negative undersampling probability to $p_n = 0.9$ gives the best overall performance. Increasing p_n leads to increased trigger recall – and this in turn enables the argument decoder to find more arguments, leading to an increase in argument classification F_1. Systems with higher trigger F_1 are not guaranteed to have better argument classification performance if they do not specifically increase trigger recall. We can show that trigger recall has a positive Pearson correlation with argument classification F_1 across all splits and p_n values. These findings are supported by the evaluation in Chapter 3 where we can show that using global decoding also leads to increased trigger recall, and this in turn leads to better argument performance.

Perpendicular to this we find that syntax encoding increases argument performance even when there is no increased trigger recall involved (for $p_n = 0$). θ_{GCN} usually performs better than θ_{treeLSTM} for finding arguments. Overall, using syntax representations leads to an increased argument recall without lowering precision too much.

We can also show the need for proper evaluation – the numbers we measure considerably fluctuate across models, regardless of data splits, p_n values, or syntax encoders. Only reporting best scores is clearly overstating a method's performance.

6 Related Work

In this chapter, we summarize the publications and methods which are most relevant to our work. Section 6.1 begins with related event extractors – we first group them based on the decoding schemes we define in Section 2.4 and discuss the resulting groups as well as the relevant publications afterwards. A special line of work if concerned with predicting triggers only – we discuss the most interesting developments there in Section 6.2. Section 6.3 touches interesting developments in general event extraction. Finally, Section 6.4 discusses syntax representations in other NLP fields.

6.1 Related Event Extractors

In the following, we want to characterize and describe the most related event extractors published before early 2018. Table 6.1 gives an overview according to our decoder classification scheme (2.4). We distinguish between local vs. global and disjoint vs. joint decoding. Local decoding incorporates information from single sentences or even shorter contexts; global decoding can draw information from the entire document or even other documents. Joint decoding refers to methods which select trigger and argument assignments jointly; disjoint decoders usually predict triggers first, and based on these decisions arguments. We will describe each point in this two-dimensional scheme, starting with local and disjoint event extractors.

6.1.1 Local and Disjoint Event Extractors

Many event extractors use local and disjoint decoding. This also includes most 'event detectors' which predict only triggers. Systems in this category are limited to single-sentence inference, and they usually first predict triggers and then arguments. Hong et al. (2011) slightly deviate from this classification because their system draws information from groups of entity mentions – decoding arguments is not strictly disjoint in

	Local	Global
Disjoint	Ahn (2006) Hong et al. (2011) Chen et al. (2015) Nguyen et al. (2016) **Chapter 4**† **Chapter 5**	
Joint	Grishman et al. (2005)* Li et al. (2013)/Li et al. (2014)	Ji and Grishman (2008)* Liao and Grishman (2010)* Yang and Mitchell (2016) **Chapter 3**

Table 6.1: Characterization matrix of event extractors. We characterize systems based on decoding scope (local vs. global) and type (joint vs. disjoint); see Section 2.4 for a discussion of this scheme. Our work is printed in bold. † marks system which predict arguments only. * marks systems which do not clearly fall in their respective categories. See their description for further details.

their case, it is more akin to argument extraction templates or entity mention distribution patterns. Nguyen et al. (2016) also use information about previous argument decisions to inform current decisions. However, they do not revise previous decisions if a better argument role assignment could be inferred. Both systems show (weak) characteristics of joint argument decoding.

6.1.1.1 Ahn (2006)

The local and disjoint pipeline in Ahn (2006) is similar to most systems in the local-disjoint class of systems: They first identify and classify triggers, and then arguments for each trigger. Ahn (2006) devise an interesting experiment: They evaluate if assigning an event type directly to words (including a 'none' type) is better than a two-stage approach where the 'triggerness' is determined before an event type is assigned. They test both approaches using a nearest-neighbor classifier (TiMBL, Daelemans et al., 2004) or a maxent classifier (MegaM, Daumé III, 2005). They find no clear performance difference between the two approaches. They find, however, that the k-nearest neighbors classifier outperforms the maxent classifier by a wide margin (about 20 F_1

points). The reason might be a bug in MegaM's bias computation[1]. Maxent classifiers are used in many event extraction publications with no reported performance issues.

Ahn (2006) sketch the (local and disjoint) system architecture which many subsequent publications used: One classifier classifies words according to their ACE event type (including ∅), and a second classifier assigns each entity mention a role (again including ∅) for each trigger in the sentence.

6.1.1.2 Hong et al. (2011)

Hong et al. (2011) are the first and only to use cross-entity information for event extraction. More precisely, they cluster all entity types into 266 subtypes; for each subtype, relevant documents for the web are retrieved, and the most important keywords are extracted from these documents. Such keywords are used to represent the centroid of each entity type cluster. Then, they apply a standard disjoint event extraction pipeline: First, triggers are predicted, then arguments, and finally the role of each argument. The entity type clusters are used to refine the argument and role decisions – the idea is to characterize an event by its event type and by the entity type clusters it usually contains in the training data. The system incorporates the information that a TRIAL event usually mentions a defendant, judge, etc. Furthermore, if the system finds that, e.g., citizens occur with terrorists as arguments of an ATTACK event, it can more easily infer that the former is the `Target` and the latter the `Attacker` of the event.

6.1.1.3 Chen et al. (2015)

To the best of our knowledge, Chen et al. (2015) are the first to use deep learning methods for event extraction. Their approach is based on Convolutional Neural Networks (Section 4.3.4). They use CNNs to capture lexical features from the contexts of triggers and arguments, and from word sequences between them. Instead of applying standard pooling of CNN outputs, they employ what they call *dynamic multi-pooling*: When they predict arguments, they take the max values from three regions of CNN outputs: left from the trigger or argument (depending on which comes first in the sentence), right from the trigger or argument, and in between. In contrast to EVENTOR, they do not use any syntax features.

[1] `http://legacydirs.umiacs.umd.edu/~hal/megam/version0_4/`

6.1.1.4 Nguyen et al. (2016)

Nguyen et al. (2016) were the first to use LSTMs (Section 4.3.3.3) for event extraction. Architecture-wise, this system is similar to EVENTOR without a syntax encoder (i.e., $\theta_{\emptyset}$). However, $\theta_{\emptyset}$ is not identical to their system. First, we use a slightly different loss. Second, we use negative undersampling, bagging, and parameter averaging. Another difference is that we do not use their 'memory matrices'. These matrices represent past decoding decisions. For example, there is an entity mention by argument role matrix. It contains an 1 whenever the system already decoded that mention to play that role, otherwise it contains 0. These matrices are fed together with the feature set from Li et al. (2013) to a final feed forward network, which combines it with information from the sentence encoder to produce final decisions. We implemented memory matrices as well, but we found no evidence that they help our system as well. [2]

6.1.2 Local and Joint Event Extractors

Local and joint systems predict event structures of an entire sentence, without crossing the sentence boundary however. The two systems we place in this category either perform joint inference by searching through a large space of (potentially multiple) event assignments for entire sentences, or they match event assignment patterns against sentences and predict the respective assignments if a match is found.

6.1.2.1 Grishman et al. (2005)

Grishman et al. (2005) present a local and (partially) joint event extractor. This is one of the earliest publications for ACE 2005. The system consists of two stages. First, it searches for matches of high precision/low recall patterns which characterize the connection between a trigger and all arguments. Second, it tries to increase recall by applying three logistic regression models after the pattern matching.

The patterns characterize the syntactic and semantic connections between triggers and arguments. One pattern type is based on the constituency head sequence between trigger and arguments, another is based on the GLARF graph (Meyers et al., 2001), a predicate-argument structure graph. Patterns are collected from the training data.

[2] Nguyen et al. (2016) is one of very few event extractors with published code (`https://github.com/anoperson/jointEE-NN`). Unfortunately, the code is not usable because it lacks critical information. There is no script published which converts the ACE data into the required input format. More severe, the actual evaluation script seems to be missing from the release.

When they are tested against test data, partial overlaps are allowed, meaning that not all arguments in the pattern need to match in the test sentence. If multiple patterns match a sentence, they are ranked by how many arguments they match, and only the best pattern is retained. Pattern matching also accounts for trigger classification – a trigger is only found if some pattern matches the current sentence. There is no dedicated trigger identification and classification stage in the system. Triggers and arguments are identified jointly. However, argument decisions are refined in a disjoint manner as described below.

Pattern matching assigns some of the entity mentions in a sentence to a potential event. All other mentions are processed by classifiers. The argument classifier predicts for a trigger-entity mention pair, if the mention is an argument of the respective event. If so, the role classifier predicts the actual role the mention plays in the event. Finally, the event classifier predicts if a trigger and a set of arguments constitute a 'presentable event' or not. The event classifier is applied to all potential events.

The sequential application of high to low precision and low to high recall components is akin to the Stanford Sieve approach (Raghunathan et al., 2010). Note that using patterns which are collected from all training documents makes brings the decoding scope more to the global side. However, decoding draws no information from other decisions it made so far. We decided against this classification to distinguish it from system which clearly perform global inference by drawing information from at least the entire current document.

The highest relevance and similarity to our work can be found in event patterns: Our dependency paths in Chapter 4 and our syntax encoders in Chapter 5 are based on a similar intuition – to grasp the syntactic relations between a trigger and its arguments. In contrast to them, however, we do not rely on immutable syntactical representations like patterns. Our representations are continuous and constitute decompositions of syntactic structures to some extent. Additionally, they can handle structures never encountered during training, while patterns can only capture information also present in the document collection they were derived from.

6.1.2.2 Li et al. (2013) and Li et al. (2014)

Li et al. (2013) and its extension Li et al. (2014) are both discussed in Chapter 3 because they constitute the base systems for IGI (the former for the setting with gold entity mentions, the latter for predicted mentions). EVENTOR (Chapter 5) uses the

static features as well. We avoid to repeat ourselves and refer the interested reader to Chapter 3 for a thorough description of Li et al. (2013) – and by extension of Li et al. (2014).

6.1.3 Global and Disjoint Event Extractors

Global and disjoint systems draw global information for improving trigger or argument decisions – they do not, however, predict triggers and arguments jointly. To put it in other words: If a disjoint base system is refined with global information, the overall system is disjoint and global; if a joint system is refined by global information, the overall system is joint and global.

We did not place any event extractor in this category. Two systems, Ji and Grishman (2008) and Liao and Grishman (2010) are candidates however because their base system (Grishman et al., 2005, Section 6.1.2.1) is not clearly joint – it uses a joint inference step, but increases argument prediction recall by an additional disjoint step. However, we decided to classify the base system as joint, because the joint aspects are more pronounced.

6.1.4 Global and Joint Event Extractors

Global and joint event extractors draw information from the entire document, or even a document collection to produce an event assignment which is globally coherent, e.g, where the same entity plays the same role in the same event.[3] However, no system performs 'true' global inference (searching a document-wide optimal event labeling), presumably because of the immense search space.

Most systems in global and joint category produce an initial (disjoint) decoding and refine it (Ji and Grishman, 2008; Liao and Grishman, 2010), or they produce perform joint decoding while enriching the local decisions with global information (IGI). Yang and Mitchell (2016) is an exception because their trigger assignment is already globally optimal.

Ji and Grishman (2008) and Liao and Grishman (2010) also refine argument decisions using global information (using a disjoint initial labeling however), while Yang and Mitchell (2016) and IGI only refine trigger assignments globally, but rely on joint inference locally.

[3] Note that such a system would ideally model event and entity coreference as well.

6.1.4.1 Ji and Grishman (2008)

Ji and Grishman (2008) build their global decoding on the 'one sense per discourse' assumption, similar to our work in Chapter 3; in contrast to all other publications we are aware of, they extend global decoding to a *collection* of topically related documents. Similar to Incremental Global Inference, they employ a base system (Grishman et al., 2005) and refine its decisions based on a list of rules with global (document-wide and cross-document-wide) scope. The rules follow two principles: Remove decisions with low confidence and refine local decisions first.

Similar to Incremental Global Inference, the global decoding in Ji and Grishman (2008) is also based on the 'one sense per discourse' assumption. However, this assumption is only a staring point for our work in Chapter 3. We also model relations between semantically related words.

Ji and Grishman (2008) start with an initial event extraction and refine the decisions based on global information. In contrast to Incremental Global Inference, the rules they employ propagate information based on hard decisions – for example, they have a rule to propagate an event type to all identical strings in a document or a collection; this propagation is made regardless of local features and base confidences. For IGI, a local decision may not be overruled by global features, either if the local features are strong indicators of an event type, or if the global feature is a rather weak indicator. Incremental Global Inference also distinguishes between different types of global information – global string match is assigned a higher weight during training than a hidden unit match. Furthermore, Ji and Grishman (2008)'s base system can only recognize event-argument patterns it learned during training whereas IGI's base system encodes more flexible information in its feature set and searches through a joint (sentence-internal) trigger-argument space. Finally, Ji and Grishman (2008) use only global information about events of the same type, whereas IGI also uses global information of heterogeneous event types.

Ji and Grishman (2008) are the first to use the widely adopted test set ('Split 1' in Chapter 5, and the test set in Chapters 3 and 4). They were also the first and (to our knowledge) only to use inter-document inference for event extraction.

6.1.4.2 Liao and Grishman (2010)

Liao and Grishman (2010) present a global and disjoint event extractor to perform easy-first inference. They use the same base system as Ji and Grishman (2008) (Grish-

man et al., 2005, Section 6.1.2.1) to get an initial event assignment for a document. Then, they use high-confidence assignments to inform other event assignments in the document, using two maximum entropy classifiers, one to propagate global trigger information, one to propagate global argument information. Their global trigger classifier is based on the observation that certain event types are more likely to occur together. Their global argument classifier is based on the observation that within a document entities play coherent roles across events. More specifically, an entity most often plays the same role in events of the same type, and it plays related roles in related events (`Target` and `Victim` in ATTACK and DIE, for example).

For Incremental Global Inference all initial event decisions are subject to global refinement, as information sources *and* as targets. Two event decisions with a low score might cause a third decision with a higher score to be reverted for example. Furthermore, our feature set is richer than the features reported in Liao and Grishman (2010): they use only (pairs of) event types and roles as features, whereas we incorporate features which also capture semantic relations.

IGI improves the entire event extraction task by improving trigger identification and classification. In contrast to Ji and Grishman (2008) and Liao and Grishman (2010), we could not successfully devise global argument features (Section 3.2.2.3).

6.1.4.3 Yang and Mitchell (2016)

Yang and Mitchell (2016)'s system is one of a few which uses predicted entity mentions. The others are Li et al. (2014) and IGI in Experiment 2, Section 3.3.2. Yang and Mitchell (2016) first decompose the task of jointly learning to predict events and entity mentions into three subtasks: learning within-event structures, learning event-event relations, and learning to extract entities. The first part corresponds to what we call joint inference. The second part corresponds to what we call global inference.

IGI with Li et al. (2014) as its base system also performs event extraction and entity mention prediction jointly. However, Yang and Mitchell (2016) go a step further: They find globally optimal assignments for both, events (with triggers and arguments) and entity mentions. They show that modeling all tasks jointly and globally improves results compared to joint and local decoding, and this in turn improves results compared to a local and disjoint system.

6.2 Event Detection

A special body of work within event extraction is concerned with predicting only triggers – a task commonly referred to as 'event detection'. Some of the reported trigger classification performances are well above the performances published for full event extractors. However, it is probably not the case that these systems, augmented with argument classification capabilities, would lead to better event extractors. We show in Section 5.7.3 that trigger recall is key to a good argument classification performance. Event detectors tend to have very high trigger classification precision and lower recall, meaning that they are probably not better suited as trigger detectors in an event detection pipeline, even if they show better trigger performance. This is also the reason why we do not compare trigger detection performances to our work.[4] In the following, we want to summarize the most interesting aspects published in this line of work in chronological order.[5]

Li et al. (2015) propose to use Abstract Meaning Representation (Banarescu et al., 2013) to improve event detection. AMR is a formalism which encodes the semantics of a sentence in a rooted, directed, and acyclic graph. They find that using AMR features (drawing information from AMR labels, node distances, and graph structure like parent nodes) improves over a baseline feature set, which is similar to the local feature set in Li et al. (2013). It seems straightforward to combine AMR with EVENTOR's syntax encoder: Instead of encoding dependency graphs, we could encode AMR graphs. This has the benefit that we do not have to characterize the semantic graph by any feature set – we could learn to represent structural information in a continuous space. We will leave this interesting idea for future research.

Bronstein et al. (2015) show that is possible to produce a reliable classifier for an event type with only a small amount of annotated triggers. The idea is very similar to the hidden unit features we present in Section 3.2.2.1: They have a small list of seed trigger words (4.2 per event type on average, collected from the ACE annotation guidelines) and build a classifier with semantic similarity and relatedness features. The classifier learns that, e.g., hyponyms of any of the seeds are likely also triggers of

[4] The 'event detection' community usually includes work from the 'event extraction' community in their contrastive evaluations. To the best of our knowledge, the opposite was never the case for publications until early 2018.

[5] Since we only include publications before 2018, and exclude most pre-prints like arXiv, we do not cover a plethora of papers which are concerned solely with event detection. It seems that this task is gaining much attention from the research community lately, especially since the deep learning paradigm shift.

the same event type. Even though Bronstein et al. (2015) report the highest trigger classification performance to date, their work has been largely ignored by the event detection community. Unfortunately, the method improves trigger precision in particular, which makes it less useful for a full event extractor.

Liu et al. (2016) use Probabilistic Soft Logic (PSL) to perform global inference of trigger assignments. PSL is similar to Markov Logic (Section 3.2.3): Soft constraints and rules can be used to search for an optimal global trigger assignment. They use a base classifier with the combined feature sets of Ahn (2006) and Hong et al. (2011) (especially the fine-grained entity types) for the base system. Global inference features include co-occurrence of triggers and LDA topics.

Finally, we want to summarize the most interesting publication which uses deep learning methods. Liu et al. (2017) introduce neural attention (Vaswani et al., 2017, i.a.) to event detection. More precisely, they work with the assumption that trigger classification can be improved if the system puts more training focus on the arguments of the respective events. Their system classifies triggers based on a word, its lexical context in a fixed-sized window, all entity types in this neighborhood, plus an attention mechanism which scores the importance of context elements. During training, they bias this scoring mechanism to assign higher scores if the words correspond to arguments of the trigger under consideration. During testing (when there are no arguments available), the system applies the same biased model to the surrounding words and entities. Liu et al. (2017) show that this in fact improves trigger classification results. Similar to Bronstein et al. (2015) this method mainly improves trigger precision.

6.3 Interesting Developments

In this Section, we present two interesting developments which were recently published. The first is a method to automatically extract large amounts of additional ACE-like training data; the other is a multi-modal event extractor which includes visual on top of textual information.

Chen et al. (2017) propose a method to automatically extract large amounts of training data. Their starting point are Compound Value Types (CVTs) in Freebase (Bollacker et al., 2008). These CVTs are mapped to ACE 2005 events, e.g., *people.marriage* is mapped to MARRY. Then, they define a centrality score for CVT arguments and ex-

tract all sentences which show all central arguments of a CVT within one sentence. All verbs which occur in as many of the *people.marriage* sentences as possible are considered triggers. Finally, all sentences which contain a *people.marriage* trigger and all corresponding central arguments are extracted as training data and mapped to the corresponding ACE structure when used for training. Only the mapping has to be done manually, all other steps involve no manual labor. Chen et al. (2017) can show that only using automatically generated training data already leads to competitive event extraction performance. It would be interesting to see how other event extractors perform with this additional training data.

Zhang et al. (2017) present a *multimodal* event extractor. Besides the ACE 2005 data, they also incorporate representations of visual information (pictures) of potential arguments into the feature set of Li et al. (2013). They use a pretrained VGG16 (Simonyan and Zisserman, 2015) model to get a numerical representation of images, and incorporate the respective vectors into the system proposed by Li et al. (2013). Zhang et al. (2017) show that visual representations greatly improve baseline event extraction results.

6.4 Syntax Representations in Other Fields

We want to discuss publications related to deep representations of dependency paths in different NLP tasks. Afterwards, we sketch other non-neural methods to decompose trees and graphs into meaningful features.

Xu et al. (2015b) and later Miwa and Bansal (2016) use dependency graphs for relation extraction. The task requires to find semantic relations between two entity mentions. Xu et al. (2015b) use shortest dependency paths and LSTMs to learn good path representations for the task at hand. This is similar to our shortest dependency path computation for event argument classification in Section 4.3.3. Instead of processing one path however, Xu et al. (2015b) split it into two parts if the path contains a change in dependency direction, i.e., if it contains a common subsumer for the two entity mentions. They use two LSTMs to process the two path parts, doubling the number of parameters in comparison to our work. They further increase the number of parameters because they use different LSTMs for different information types: words, part-of-speech tags, dependency relations and WordNet hypernyms. Our bidirectional LSTM encodes most of this information (excluding hypernyms, but includ-

ing entity types along the way) in one pass, meaning that the representation of a word like 'we' is different if it is a subject or an object of a verb, for example.

Miwa and Bansal (2016) extend the representation from multiple LSTMs for left and right path parts and different information channels to one tree which contains both entity mentions which are involved in a semantic relation as end points. Similar to our θ_{treeLSTM} syntax encoder, their tree LSTM is an extension of the child-sum tree LSTM (Tai et al., 2015) to a bidirectional tree LSTM with dependency labels along the tree. In contrast to our work, they use label-dependent weight tensors, while we use dependency label embeddings to encode syntax labels (Section 5.5.2.3). Label-dependent weight tensors are equivalent to one weight matrix per dependency label – such tensors are much bigger than standard LSTM weight matrices. Our dependency embeddings only introduce $d \times e$ new parameters, where d is the number of dependency labels and e the embeddings dimensionality.

Roth and Lapata (2016) use LSTM representations of shortest dependency paths for Semantic Role Labeling (Section 2.3.3). Similar to our work in Section 4.3.3, they fix the endpoint types of their paths: predicates always come first, potential arguments last. However, they only use the last hidden state of the LSTM to represent the entire path, while we always forward all hidden states for final classification, relieving the burden for the LSTM to encode everything into one final vector.

Marcheggiani and Titov (2017) also work on Semantic Role Labeling, but they operate on the entire dependency graph. For this, they use Graph Convolutional Networks. We discuss their work when we introduce θ_{GCN} (Section 5.5.3).

Finally, we want to mention two non-neural methods which make trees (or graphs in general) accessible to Support Vector Machines (Hearst et al., 1998), namely Tree Kernels (Culotta and Sorensen, 2004) and Graph Kernels (Gärtner et al., 2003). Both work similarly: They define a similarity function between two trees/graphs essentially by counting how many substructures (subtrees/subgraphs) of varying length they have in common. While this operation can be implemented efficiently for trees, it is inefficient for general graphs. Such kernels can be used in combination with Support Vector Machines to directly operate on trees/graphs without the necessity to translate them into shallow features, similarly to treeLSTMs.

7 Conclusions and Future Work

We want to end this thesis by summarizing our most important contributions, and by giving an outlook into interesting future research.

7.1 Conclusions

ACE casts event extraction as an information extraction task. First, we have to identify triggers – words which most clearly indicate the presence of an event. For each trigger, we have to identify its arguments, that is, the entities playing some role in the event. The overall task is demanding. There is a strong inter-dependency between trigger and argument detection – missed triggers imply missed arguments, spurious triggers imply spurious arguments.

Furthermore, ACE triggers are scarce: Only 1.3% of words in the widely used training set are event triggers. In similar tasks like Semantic Role Labeling or Frame-semantic Parsing, the percentage of words which bear categories of interest is much larger. However, many triggers are strong indicators of their events, e.g, the words 'attack' or 'marry'. Most systems heavily rely on lexical features for trigger identification and classification. For example, most event detection (trigger-only) systems rely exclusively on word embeddings and elaborate deep learning architectures to identify and classify triggers.

In Chapter 3, we investigate if and how the global, document-wide context can improve a local and joint event extractor. We find that we can greatly improve trigger recall by incorporating global (document-wide) information about semantically related words. We refine the local (intra-sentential) trigger decisions of a system which predicts triggers and arguments jointly by drawing information about semantically related words which were classified as event triggers somewhere in the document.

In this thesis, we also show that improving argument identification and classification performance is challenging. First, it does not suffice to claim that a method improves

argument performance by showing that the respective evaluation numbers improved. As we show in Chapter 3, and again in Chapter 5, improving trigger classification recall has a high chance to result in better argument performance, without the necessity to actually improve argument performance methodologically. It seems crucial for a good overall event extraction performance to have a high trigger recall. Trigger precision is of course also important – systems which assigns some event type to every eligible word do not have a good argument performance because they would introduce too many spurious arguments. Nevertheless, we find trigger recall to be more important, given that the system has a reasonable trigger precision.

Because of this strong interdependency between trigger and argument predictions, and because we cannot devise effective global argument features (Section 3.2.2.3), we analyze argument identification and classification performance in isolation (Chapter 4) and set all trigger decisions to their gold value. We perform the investigation with our base system from Chapter 3 and find that performance rapidly drops with increasing distance of an argument to its trigger. Performance loss is about 10 F_1 points per additional dependency edge. This observation is the foundation for our work in Chapters 4 and 5. Our idea is to devise methods which operate on syntax structures to better handle long-range dependencies between triggers and arguments.

We propose a deep learning architecture which (1) learns to identify which parts of dependency paths conveying useful information and (2) deals with unseen paths (Chapter 4). Both points are insufficiently handled by most event extractors, which rely on categorical features or on simple (direct) dependency relations. Using simple, short-range (mostly direct) dependency relations is problematic because they cannot capture long-range dependencies of arguments which are farther away. Some argument types tend to be expressed more distant to their triggers than others, which leads to a worse performance when predicting them. Our methods encode entire dependency paths, or even dependency graphs, to represent as much syntax information as needed. We can show that Long Short-Term Memory networks operating on lexicalized dependency paths increase argument performance, especially on paths of lengths 1-3.[1]

This finding is again validated in Chapter 5, where we use syntax encoders on dependency graphs for full event extraction. Chapter 5 complements our investigation of syntax representations in particular, and of event extraction with deep learning methods in general. The chapter is dedicated to three major questions: Do the results in

[1]Longer dependency paths are scarce, leading to unreliable evaluation numbers.

Chapter 4 also hold 'in the wild', that is, with predicted and noisy triggers? Does syntax encoding also benefit trigger prediction? And finally, how can we derive a scientifically reliable answer to the previous questions?

First, we propose a base system for event extraction which is similar to a state-of-the-art 'neural' event extractor. The base system is on the one hand our baseline, on the other a common framework for the two syntax encoders we use. A common framework addresses the first point to ensure the best evaluation reliability (systems have to use the same preprocessing, Section 2.5.1). The second point (indeterministic training procedures lead to significantly better evaluation results by pure luck, Section 2.5.2) is addressed by our strategy to always train five models and average their performances for all evaluation measures. We use this procedure every time we use deep learning methods (Chapters 4 and 5). This gives us a better performance estimate and some quantification of the error involved in every evaluation measure. We can show (Appendix A.5) that the evaluation procedure described above is necessary: F_1 score fluctuations of 1 point are common for all methods and data splits. In some cases, fluctuations are 2 F_1 points or higher. Given that published event extraction numbers are quite close together, a 2 F_1 points performance increase may already constitute a new state-of-the-art.

The general idea of our system in Chapter 5 is to encode the entire dependency structure of a sentence along with its lexical content, and to derive better event decisions based on this information. On top of the base system, we use Graph Convolutional Networks and tree-shaped Long Short-Term Memory networks as syntax encoders. The first focuses on the local dependency relations of a word. The latter produces a more global representation where each node in the dependency graph encodes syntactic relations to all its children and to the root node, and the root node encodes the entire graph.[2] We show that argument identification and classification benefits from higher-order syntax representations even with predicted and noisy triggers. Trigger identification and classification however seems to benefit only marginally from it. We suspect that for triggers and most important syntax information is already present in their local dependency relations, since many triggers are verbs. In terms of syntax encoders, we find a slight benefit of Graph Convolutional Networks, presumably because they have less parameters, which complies with the low amount of training data in ACE 2005.

[2]For simplicity, we compute our dependency graph representations on a minimum spanning tree of the graph, rooted at the dependency root node.

Besides more complex syntax encoders, we also investigate methods to better handle class imbalance and scarce training data in Chapter 5. The first problem is mitigated by a new undersampling method we propose (Section 5.6.2), named repeated negative undersampling. The idea is simple: We avoid to train on a non-trigger word with a certain probability. A higher probability leads to more triggers which the system sees during training compared to non-triggers, effectively biasing the system towards paying more attention to trigger prediction and increasing trigger recall. Our main contribution however is the observation that it is beneficial to re-evaluate the undersampled non-trigger words anew during training, meaning that the system is exposed to *different* non-triggers, but to only a small amount of them in each training epoch. This strategy considerably increases trigger recall for all methods we evaluate, and this in turn increases also argument performance. The other method we investigate is bagging. Bagging constructs n different versions of the training data by withholding a certain amount of data points from the sampling procedure – we use these points as a development set. Then, bagging trains n classifiers (optimized on n development sets) and averages their predictions during testing. Bagging has a positive effect when training data is scarce – this was shown for a multitude of tasks and data sets. We can also show a positive effect on event extraction and the ACE data.

7.2 Future Work

We outlined future work throughout the thesis. Here, we want to collect the most important suggestions for future research. This section will conclude the thesis.

The most obvious research line is the combination of Incremental Global Inference with the syntax encoders we present in Chapter 5. To the best of our knowledge, there is no global and deep event extractor published to date. IGI *as a method* can work with any base system. The idea is to inform a neural, intra-sentential base system about semantically related words in the document in an incremental way. We suspect that the resulting event extractor would be superior in terms of performance. IGI *as a concrete implementation* however is tailored towards structured perceptrons. More work is needed to adapt it to gradient optimization training.

Another line of work we suggest for future work is concerned with improving the local event extractor in general. In Section 5.6.1, we discuss the limitations of our loss function, mainly its assumption that trigger and argument decisions are inde-

pendent. This assumption is clearly wrong. To model the inter-dependency between triggers and arguments however, one has to change the loss function to a globally normalized version, e.g., using the conditional random field (CRF) loss described in Andor et al. (2016). CRFs model sequences in terms of transition probabilities, making certain transitions very unlikely while promoting others. CRFs proved useful for many sequence prediction tasks in NLP, e.g., part-of-speech tagging and named entity recognition. We think however that the standard linear chain CRF is not enough for event extraction (even though is a good first approximation); a structured CRF is more appropriate to model the interdependencies between *all* arguments and a potential trigger. The system would not compute softmax distributions for a dependency graph, but CRF probability distributions, which take the entire graph structure into account. This would have the effect that spurious double or triple assignments of roles occur less often.

A last suggestion targets the structure of our syntax encoders: The syntax itself. We already placed event extraction as a task into the vicinity of semantic parsing tasks like Semantic Role Labeling and Frame-Semantic Parsing (Section 2.3.3). An interesting line of research in event extraction is to encode *semantic structures* instead or on top of syntactic structures. Semantic structures can be readily processed with the methods we describe in chapter 5. Semantic Role Labeling for example produces a tree-like structure with the predicate as the root and its arguments as children. More complex semantic structures like AMR (Section 6.2) could also be readily processed. The new semantic encoders would produce valuable information for event extraction.

We also believe that event extraction and semantic parsing tasks can inform each other – a perfect training regime would be a multi-task setting where one system predicts both, events and predicate-argument structures. However, no such dataset is currently available. Another strategy could be to pre-train an event extractor on predicate-argument structures, before training it on the ACE data (or vice versa).

List of Figures

List of Tables

Bibliography

ACE2005. 2005. The ACE 2005 (ACE 05) evaluation plan. https://pdfs.semanticscholar.org/ae48/278ccc098d49b5c9dbe695f466c540f42df7.pdf.

David Ahn. 2006. The stages of event extraction. In *Proceedings of the Workshop on Annotating and Reasoning about Time and Events,* Sydney, Australia, 23 July 2006, pages 1–8.

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers),* Berlin, Germany, 7–12 August 2016, pages 2442–2452. Association for Computational Linguistics.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse,* Sofia, Bulgaria, 8-9 August, pages 178–186.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1993. The problem of learning long-term dependencies in recurrent networks. In *1993 Neural Networks for Computing Conference*, Utah, USA.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5 (2):157–166.

James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305.

Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: A collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data,* Vancouver, B.C., Canada, 10–12 June 2008, pages 1247–1250.

Leo Breiman. 1996. Bagging predictors. *Machine learning*, 24(2):123–140.

Ofer Bronstein, Ido Dagan, Qi Li, Heng Ji, and Anette Frank. 2015. Seed-based event trigger labeling: How far can event descriptions get us? In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers),* Beijing, China, 26–31 July 2015, volume 2, pages 372–376.

Peter F. Brown, Vincent J. Della Pietra, Peter V. DeSouza, Jenifer C. Lai, and Robert L. Mercer. 1992. Class-based n-gram models of natural language. *Computational Linguistics*, 18(4):467–479.

Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357.

Yubo Chen, Liheng Xu, Kang Liu, Daojian Zeng, and Jun Zhao. 2015. Event extraction via dynamic multi-pooling convolutional neural networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers),* Beijing, China, 26–31 July 2015, pages 167–176.

Yubo Chen, Shulin Liu, Xiang Zhang, Kang Liu, and Jun Zhao. 2017. Automatically labeled data generation for large scale event extraction. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers),* Vancouver, B.C., Canada, 30 July –4 August 2017, pages 409–419. Association for Computational Linguistics.

François Chollet et al. 2015. Keras. `https://github.com/fchollet/keras`.

Michael Collins. 2002. Discriminative training methods for Hidden Markov Models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing,* Philadelphia, Penn., 6–7 July 2002, pages 1–8.

Sven F Crone, Stefan Lessmann, and Robert Stahlbock. 2006. The impact of preprocessing on data mining: An evaluation of classifier sensitivity in direct marketing. *European Journal of Operational Research*, 173(3):781–800.

Aron Culotta and Jeffrey Sorensen. 2004. Dependency tree kernels for relation extraction. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics,* Barcelona, Spain, 21–26 July 2004, pages 423–429.

Walter Daelemans, Jakub Zavrel, Ko van der Sloot, and Antal van den Bosch. 2004. TiMBL: Tilburg Memory Based Learner, version 5.1, Reference Guide. Technical Report ILK 04-02, ILK Tilburg.

Dipanjan Das, Desai Chen, André F. T. Martins, Nathan Schneider, and Noah A. Smith. 2014. Frame-semantic parsing. *Computational Linguistics*, 40(1):9–56.

Hal Daumé III. 2005. Mega model optimization package. `http://users.umiacs.umd.edu/~hal/megam/version0_3/`.

Donald Davidson. 1969. The individuation of events. In *Essays in honor of Carl G. Hempel*, pages 216–234. Springer.

Marie-Catherine De Marneffe and Christopher D Manning. 2008. Stanford typed dependencies manual. Technical report, Technical report, Stanford University.

David Dowty. 1991. Thematic proto-roles and argument selection. *language*, 67(3): 547–619.

Timothy Dozat. 2016. Incorporating nesterov momentum into adam.

Bradley Efron. 1992. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*, pages 569–593. Springer.

Charles Elkan. 2001. The foundations of cost-sensitive learning. In *International joint conference on artificial intelligence*, volume 17, pages 973–978. Lawrence Erlbaum Associates Ltd.

Enhanced Dependencies Website. 2017. Enhanced dependencies in ud v2. `http://universaldependencies.org/v2/enhanced.html`. Accessed:2018-05-29.

Christiane Fellbaum, editor. 1998. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, Mass.

Xiaocheng Feng, Lifu Huang, Duyu Tang, Bing Qin, Heng Ji, and Ting Liu. 2016. A language-independent neural network for event detection. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers),* Berlin, Germany, 7–12 August 2016, page 66.

Charles Fillmore. 1982. Frame semantics. *Linguistics in the morning calm*.

Charles J. Fillmore, Christopher R. Johnson, and Miriam R. L. Petruck. 2003. Background to FrameNet. *International Journal of Lexicography*, 16(3):235–250.

Yoav Freund and Robert Shapire. 1999. A short introduction to boosting. *Journal of the Japanese Society for Artificial Intelligence*, 14:771–780.

William A. Gale, Kenneth W. Church, and David Yarowsky. 1992. One sense per discourse. In *Proceedings of the DARPA Speech and Natural Language Workshop,* New York, N.Y., 23–26 February 1992, pages 233–237.

Thomas Gärtner, Peter Flach, and Stefan Wrobel. 2003. On graph kernels: Hardness results and efficient alternatives. In *Learning theory and kernel machines*, pages 129–143. Springer.

Daniel Gildea and Daniel Jurafsky. 2002. Automatic labeling of semantic roles. *Computational Linguistics*, 28(3):245–288.

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics,* Sardinia, Italy, May 13-15, pages 249–256.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*.

Ralph Grishman, David Westbrook, and Adam Meyers. 2005. NYU's English ACE 2005 system description. Technical report, Department of Computer Science, New York University, New York, N.Y.

Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. 1998. Support vector machines. *IEEE Intelligent Systems and their applications*, 13 (4):18–28.

GE Hinton, JL McClelland, and DE Rumelhart. 1986. Distributed representations. In *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1*, pages 77–109. MIT Press.

Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Yu Hong, Jianfeng Zhang, Bin Ma, Jianmin Yao, Guodong Zhou, and Qiaoming Zhu. 2011. Using cross-entity inference to improve event extraction. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers),* Portland, Oreg., 19–24 June 2011, pages 1127–1136.

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies,* Montréal, Québec, Canada, 3–8 June 2012, pages 142–151.

Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456.

Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An introduction to statistical learning*, volume 112. Springer, New York.

Heng Ji and Ralph Grishman. 2008. Refining event extraction through cross-document inference. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies,* Columbus, Ohio, 15–20 June 2008, pages 254–262.

Rie Johnson and Tong Zhang. 2015. Effective use of word order for text categorization with Convolutional Neural Networks. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies,* Denver, Col., 31 May – 5 June 2015, pages 103–112.

Alex Judea. 2021a. Accompanying Code for Chapter 4 of the PhD Thesis "Global Inference and Local Syntax Representations for Event Extraction". doi: 10.11588/data/CZZEKX.

Alex Judea. 2021b. Accompanying Code and Models for Chapter 5 of the PhD Thesis "Global Inference and Local Syntax Representations for Event Extraction". doi: 10.11588/data/Z1RKOI.

Alex Judea and Michael Strube. 2015. Event extraction as frame-semantic parsing. In *Proceedings of STARSEM 2015: The Fourth Joint Conference on Lexical and Computational Semantics,* Denver, Col., 4–5 June 2015, pages 159–164.

Alex Judea and Michael Strube. 2016. Incremental global event extraction. In *Proceedings of the 26th International Conference on Computational Linguistics,* Osaka, Japan, 11–16 December 2016, pages 2279–2289.

Alex Judea and Michael Strube. 2017. Event argument identification on dependency graphs with bidirectional lstms. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing,* Taipei, Taiwan, 27 November–1 December 2017, pages 822–831.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On large-batch training for deep learning: Generalization gap and sharp minima. *Proceedings of the Fifth International Conference on Learning Representations,* Toulon, France, 24 April–26 April 2017.

Jin-Dong Kim, Tomoko Ohta, Sampo Pyysalo, Yoshinobu Kano, and Jun'ichi Tsujii. 2009. Overview of bionlp'09 shared task on event extraction. In *Proceedings of the Workshop on Current Trends in Biomedical Natural Language Processing: Shared Task,* Boulder, Colorado — June 04 - 05, pages 1–9. Association for Computational Linguistics.

Yoon Kim. 2014. Convolutional Neural Networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing,* Doha, Qatar, 25–29 October 2014, pages 1746–1751.

Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *Proceedings of the Fifth International Conference on Learning Representations,* Toulon, France, 24 April–26 April 2017.

Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. 2012. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer.

Qi Li, Heng Ji, and Liang Huang. 2013. Joint event extraction via structured prediction with global features. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers),* Sofia, Bulgaria, 4–9 August 2013, pages 73–82.

Qi Li, Heng Ji, Yu Heng, and Sujian Li. 2014. Constructing information networks using one single model. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing,* Doha, Qatar, 25–29 October 2014, pages 1846–1851.

Xiang Li, Thien Huu Nguyen, Kai Cao, and Ralph Grishman. 2015. Improving event detection with abstract meaning representation. In *Proceedings of the First Workshop on Computing News Storylines,* Beijing, China, 31 July, pages 11–15.

Shasha Liao and Ralph Grishman. 2010. Using document level cross-event inference to improve event extraction. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics,* Uppsala, Sweden, 11–16 July 2010, pages 789–797.

Linguistic Data Consortium: Entities. 2005. Ace (automatic content extraction) english annotation guidelines for entities.

Linguistic Data Consortium: Events. 2005. Ace (automatic content extraction) english annotation guidelines for events.

Jiangming Liu and Yue Zhang. 2017. Encoder-decoder shift-reduce syntactic parsing. In *Proceedings of the 15th International Conference on Parsing Technologies,* Pisa, Italy, September 20-22, pages 105–114. Association for Computational Linguistics.

Shaobo Liu, Rui Cheng, Xiaoming Yu, and Xueqi Cheng. 2018. Exploiting contextual information via dynamic memory network for event detection. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing,* Brussels, Belgium, 2–4 November 2018.

Shulin Liu, Kang Liu, Shizhu He, and Jun Zhao. 2016. A probabilistic soft logic based approach to exploiting latent and global information in event classification. In *AAAI*, pages 2993–2999.

Shulin Liu, Yubo Chen, Kang Liu, and Jun Zhao. 2017. Exploiting argument information to improve event detection via supervised attention mechanisms. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers),* Vancouver, B.C., Canada, 30 July –4 August 2017, volume 1, pages 1789–1798.

Xu-Ying Liu, Jianxin Wu, and Zhi-Hua Zhou. 2009. Exploratory undersampling for class-imbalance learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(2):539–550.

Qing Lu and Lise Getoor. 2003. Link-based classification. In *Proceedings of the 20th International Conference on Machine Learning,* Washington, D.C., 21–24 August 2003, pages 496–503.

Catherine Macleod, Ralph Grishman, Adam Meyers, Leslie Barrett, and Ruth Reeves. 1998. Nomlex: A lexicon of nominalizations. In *Proceedings of EURALEX*, volume 98, pages 187–193.

Christopher D. Manning and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, Mass.

Diego Marcheggiani and Ivan Titov. 2017. Encoding sentences with graph convolutional networks for semantic role labeling. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing,* Copenhagen, Denmark, 7–11 September 2017, pages 1507–1516. Association for Computational Linguistics.

Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The Penn treebank: Annotating predicate argument structure. In *Proceedings of ARPA Speech and Natural Language Workshop*.

Lluís Màrquez, Xavier Carreras, Kenneth C Litkowski, and Suzanne Stevenson. 2008. Semantic role labeling: an introduction to the special issue.

Sebastian Martschat and Michael Strube. 2015. Latent structures for coreference resolution. *Transactions of the Association for Computational Linguistics*, 3:405–418.

Adam Meyers, Michiko Kosaka, Satoshi Sekine, Ralph Grishman, and Shubin Zhao. 2001. Parsing and glarfing. In *Proceedings of RANLP-2001, Recent Advances in Natural Language Processing, Tzigov Chark, Bulgaria, 5-7 September*.

Scott Miller, Jethran Guinness, and Alex Zamanian. 2004. Name tagging with word clusters and discriminative training. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics,* Boston, Mass., 2–7 May 2004.

Makoto Miwa and Mohit Bansal. 2016. End-to-end relation extraction using lstms on sequences and tree structures. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers),* Berlin, Germany, 7–12 August 2016, pages 1105–1116. Association for Computational Linguistics.

Makoto Miwa, Paul Thompson, Ioannis Korkontzelos, and Sophia Ananiadou. 2014. Comparable study of event extraction in newswire and biomedical domains. In *Proceedings of the 25th International Conference on Computational Linguistics,* Dublin, Ireland, 23–29 August 2014, pages 2270–2279.

Marc Moens and Marc Steedman. 1988. Temporal ontology and temporal reference. *Computational Linguistics*, 14(2):15–28.

Alexander PD Mourelatos. 1978. Events, processes, and states. *Linguistics and philosophy*, 2(3):415–434.

Michael C Mozer. 1995. A focused backpropagation algorithm for temporal. *Backpropagation: Theory, architectures, and applications*, page 137.

Jennifer Neville and David Jensen. 2000. Iterative classification in relational data. In *Proceedings of the AAAI-2000 Workshop on Learning Statistical Models from Relational Data,* Austin, Texas, July 31, pages 13–20. AAAI Press.

Thien Huu Nguyen, Kyunghyun Cho, and Ralph Grishman. 2016. Joint event extraction via recurrent neural networks. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies,* San Diego, Cal., 12–17 June 2016, pages 300–309.

Eric W. Noreen. 1989. *Computer Intensive Methods for Hypothesis Testing: An Introduction*. Wiley, New York, N.Y.

Martha Palmer, Daniel Gildea, and Paul Kingsbury. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31(1):71–105.

Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies,* New Orleans, Louisiana, 1–6 June 2018, volume 1, pages 2227–2237.

Hoifung Poon and Lucy Vanderwende. 2010. Joint inference for knowledge extraction from biomedical literature. In *Proceedings of Human Language Technologies 2010: The Conference of the North American Chapter of the Association for Computational Linguistics,* Los Angeles, Cal., 2–4 June 2010, pages 813–821. Association for Computational Linguistics.

James Pustejovsky. 1991. The generative lexicon. *Computational Linguistics*, 17(4): 209–241.

James Pustejovsky, José M Castano, Robert Ingria, Roser Sauri, Robert J Gaizauskas, Andrea Setzer, Graham Katz, and Dragomir R Radev. 2003a. Timeml: Robust specification of event and temporal expressions in text. *New directions in question answering*, 3:28–34.

James Pustejovsky, Patrick Hanks, Roser Sauri, Andrew See, Robert Gaizauskas, Andrea Setzer, Dragomir Radev, Beth Sundheim, David Day, Lisa Ferro, et al. 2003b. The timebank corpus. In *Corpus linguistics*, volume 2003, page 40. Lancaster, UK.

Karthik Raghunathan, Heeyoung Lee, Sudarshan Rangarajan, Nathanael Chambers, Mihai Surdeanu, Dan Jurafsky, and Christopher Manning. 2010. A multi-pass sieve for coreference resolution. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing,* Cambridge, Mass., 9–11 October 2010, pages 492–501.

Lev Ratinov and Dan Roth. 2009. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning,* Boulder, Colorado, June 4, pages 147–155. Association for Computational Linguistics.

Nils Reimers and Iryna Gurevych. 2017. Reporting score distributions makes a difference: Performance study of lstm-networks for sequence tagging. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing,* Copenhagen, Denmark, 7–11 September 2017, pages 338–348. Association for Computational Linguistics.

Nils Reimers and Iryna Gurevych. 2018. Why comparing single performance scores does not allow to draw conclusions about machine learning approaches. *CoRR*, abs/1803.09578.

Matthew Richardson and Pedro Domingos. 2006. Markov logic networks. *Machine Learning*, 62(1-2):107–136.

Sebastian Riedel, Hong-Woo Chun, Toshihisa Takagi, and Jun'ichi Tsujii. 2009. A markov logic approach to bio-molecular event extraction. In *Proceedings of the Workshop on Current Trends in Biomedical Natural Language Processing: Shared Task,* Boulder, Colorado, June 5, pages 41–49. Association for Computational Linguistics.

Michael Roth and Mirella Lapata. 2016. Neural semantic role labeling with dependency path embeddings. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers),* Berlin, Germany, 7–12 August 2016, pages 1192–1202.

Sebastian Schuster and Christopher D. Manning. 2016. Enhanced english universal dependencies: An improved representation for natural language understanding tasks. In *Proceedings of the 10th International Conference on Language Resources and Evaluation,* Portorož, Slovenia, 23–28 May 2016.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI Magazine*, 29(3): 93–106.

K. Simonyan and A. Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the Third International Conference on Learning Representations,* San Diego, CA., May 7-9 2015.

Samuel L Smith and Quoc V Le. 2018. A bayesian perspective on generalization and stochastic gradient descent. *Proceedings of the Sixth International Conference on Learning Representations,* Vancouver, Canada, 30 April–3 May 2018.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing,* Seattle, Wash., 18–21 October 2013, pages 1631–1642.

Zhiyi Song, Ann Bies, Stephanie Strassel, Tom Riese, Justin Mott, Joe Ellis, Jonathan Wright, Seth Kulick, Neville Ryant, and Xiaoyi Ma. 2015. From light to rich ere: annotation of entities, relations, and events. In *Proceedings of the 3rd Workshop on EVENTS at the Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 89–98.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

Ang Sun, Ralph Grishman, and Satoshi Sekine. 2011. Semi-supervised relation extraction with large-scale word clustering. In *Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies*, pages 521–529.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers),* Beijing, China, 26–31 July 2015, pages 1556–1566. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

Marc Verhagen, Robert Gaizauskas, Frank Schilder, Mark Hepple, Graham Katz, and James Pustejovsky. 2007. Semeval-2007 task 15: Tempeval temporal relation identification. In *Proceedings of the 4th international workshop on semantic evaluations,* Prague, Czech Republic — June 23 - 24, pages 75–80. Association for Computational Linguistics.

Marc Verhagen, Roser Sauri, Tommaso Caselli, and James Pustejovsky. 2010. Semeval-2010 task 13: Tempeval-2. In *Proceedings of the 5th international workshop on semantic evaluation,* Los Angeles, California — July 15 - 16, pages 57–62. Association for Computational Linguistics.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning,* Montreal, QC, Canada, June 14 - 18, pages 1113–1120. ACM.

Wikipedia contributors. 2018. Murmurhash — Wikipedia, the free encyclopedia. [Online; accessed 19-June-2018].

Yan Xu, Lili Mou, Ge Li, Yunchuan Chen, Hao Peng, and Zhi Jin. 2015a. Classifying relations via long short term memory networks along shortest dependency paths. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing,* Lisbon, Portugal, 17–21 September 2015, pages 1785–1794.

Yan Xu, Lili Mou, Ge Li, Yunchuan Chen, Hao Peng, and Zhi Jin. 2015b. Classifying relations via Long Short Term Memory Networks along shortest dependency paths. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing,* Lisbon, Portugal, 17–21 September 2015, pages 1785–1794.

Bishan Yang and Tom Mitchell. 2016. Joint extraction of events and entities within a document context. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies,* San Diego, Cal., 12–17 June 2016, pages 289–299. Association for Computational Linguistics.

Tongtao Zhang, Spencer Whitehead, Hanwang Zhang, Hongzhi Li, Joseph Ellis, Lifu Huang, Wei Liu, Heng Ji, and Shih-Fu Chang. 2017. Improving event extraction via multimodal integration. In *Proceedings of the 2017 ACM on Multimedia Conference,* Mountain View, CA, USA, October 23 - 27, pages 270–278. ACM.

A Appendix

A.1 Allowed Trigger Parts-of-speech

As mentioned in Section 3.1, a trigger label is only assignable to words which match the following regular expression:

"IN|JJ|RB|DT|VBG|VBD|NN|NNPS|VB|VBN|NNS|VBP|NNP|PRP|VBZ"

The part-of-speech tags are the Penn Treebank tags Marcus et al. (1994).

A.2 Chapter 3: Base System Features

Table A.1 presents features of the base system we use in Chapter 3. The argument features are also used for the event extractor in Chapter 5.

A.3 Confusion Heat Map for Incremental Global Inference's Base System

Figure A.1 presents a heat map for Incremental Global Inference's base system (Chapter 3).

A.4 Event Argument Classification Evaluation

Table A.3 presents the full version of Table 4.3.

Category	Type	Feature Description
Trigger	Lexical	1. unigrams/bigrams of the current and context words within the window of size 2 2. unigrams/bigrams of part-of-speech tags of the current and context words within the window of size 2 3. lemma and synonyms of the current token 4. base form of the current token extracted from Nomlex (Macleod et al., 1998) 5. Brown clusters that are learned from ACE English corpus (Brown et al., 1992; Miller et al., 2004; Sun et al., 2011). We used the clusters with prefixes of length 13, 16 and 20 for each token.
	Syntactic	6. dependent and governor words of the current token 7. dependency types associated the current token 8. whether the current token is a modifier of job title 9. whether the current token is a non-referential pronoun
	Entity Information	10. unigrams/bigrams normalized by entity types 11. dependency features normalized by entity types 12. nearest entity type and string in the sentence/clause
Argument	Basic	1. context words of the entity mention 2. trigger word and subtype 3. entity type, subtype and entity role if it is a geo-political entity mention 4. entity mention head, and head of any other name mention from co-reference chain 5. lexical distance between the argument candidate and the trigger 6. the relative position between the argument candidate and the trigger: {before, after, overlap, or separated by punctuation} 7. whether it is the nearest argument candidate with the same type 8. whether it is the only mention of the same entity type in the sentence
	Syntactic	9. dependency path between the argument candidate and the trigger 10. path from the argument candidate and the trigger in constituent parse tree 11. length of the path between the argument candidate and the trigger in dependency graph 12. common root node and its depth of the argument candidate and parse tree 13. whether the argument candidate and the trigger appear in the same clause

Table A.1: Static feature templates of our base system used in Chapter 3. The argument features are also used in Chapter 5. This table is an exact reproduction of the local feature table in Li et al. (2013).

Figure A.1: A heat map (darker colors mean higher values) representation of the confusion matrix for Incremental Global Inference's base system. Gold labels are on the Y axis, predicted on the X axis.

Category	Feature Description
Trigger	1. bigram of trigger types occur in the same sentence or the same clause 2. binary feature indicating whether synonyms in the same sentence have the same trigger label 3. context and dependency paths between two triggers conjuncted with their types
Argument	4. context and dependency features about two argument candidates which share the same role within the same event mention 5. features about one argument candidate which plays as arguments in two event mentions in the same sentence 6. features about two arguments of an event mention which are overlapping 7. the number of arguments with each role type of an event mention conjuncted with the event subtype 8. the pairs of time arguments within an event mention conjuncted with the event subtype

Table A.2: Dynamic feature templates of our base system used in Chapter 3. This table is a reproduction of the global feature table in Li et al. (2013).

A.5 Chapter 5: All Evaluation Tables

Here, we present all evaluation numbers of all runs we otherwise present in aggregated form (mean evaluation numbers with standard deviations) in Chapter 5. The following tables report all numbers for data splits 1, 2, and 3.

A.5.1 Split 1

Tables A.4-A.12 report evaluation numbers for $\theta_{\emptyset}$ (no syntax), θ_{GCN}, and θ_{treeLSTM} for Split 1 and p_n values 0.0, 0.5, and 0.9.

A.5.2 Split 2

Tables A.13-A.21 report evaluation numbers for $\theta_{\emptyset}$ (no syntax), θ_{GCN}, and θ_{treeLSTM} for Split 2 and p_n values 0.0, 0.5, and 0.9.

A.5.3 Split 3

Tables A.22-A.30 report evaluation numbers for $\theta_{\emptyset}$ (no syntax), θ_{GCN}, and θ_{treeLSTM} for Split 3 and p_n values 0.0, 0.5, and 0.9.

A.6 Chapter 5: Average Significance

Table A.31 reports statistical significance scores (lower is better) for the respective F_1 trigger or argument score of θ_{GCN} and θ_{treeLSTM} compared to $\theta_{\emptyset}$. We have five models for θ_{GCN}, θ_{treeLSTM}, and $\theta_{\emptyset}$. Before we compute significance, we average the performance scores per sentence across the five models and then we compute significance using approximate randomization (Noreen, 1989). This procedure is a very strict measure because it averages individual strong models out. Normally, scores are regarded significant if $p < 0.05$. In this case however, higher p values might also be regarded significant, although we cannot make a guess for a concrete threshold.

A.7 Code and Data Used in this Thesis

The code and data used in this thesis has been published with the following digital object identifiers (DOI):

- Alex Judea. 2021a. Accompanying Code for Chapter 4 of the PhD Thesis "Global Inference and Local Syntax Representations for Event Extraction". doi: 10.11588/data/CZZEKX. URL `https://doi.org/10.11588/data/Z1RKOI`
- Alex Judea. 2021b. Accompanying Code and Models for Chapter 5 of the PhD Thesis "Global Inference and Local Syntax Representations for Event Extraction". doi: 10.11588/data/Z1RKOI. URL `https://doi.org/10.11588/data/CZZEKX`

	Baseline			EVENTOR				
	P	R	F_1	P	R	F_1	$\Delta F_1 \pm \sigma$	Support
Micro	67.7	58.7	62.9	63.1	68.2	65.5†	2.7 ± 0.5	916
no CNN	67.7	58.7	62.9	66.2	62.5	64.2†	1.3 ± 0.8	916
Time	69.9	70.9	70.4	70.9	80.1	75.2	4.8 ± 1.2	134
Entity	63.7	56.7	60.0	57.7	65.2	61.2	1.2 ± 2.9	127
Place	64.0	41.7	50.5	52.1	48.0	49.9	-0.6 ± 1.7	115
Person	74.6	61.7	67.6	69.1	78.3	73.4	5.8 ± 2.1	81
Artifact	78.5	71.8	75.0	70.3	77.2	73.5	-1.5 ± 1.2	71
Destination	63.4	66.7	65.0	65.6	80.0	72.1	7.1 ± 1.0	39
Crime	84.4	100.0	91.6	82.5	99.5	90.2	-1.3 ± 0.6	38
Attacker	60.7	47.2	53.1	52.4	66.6	58.6	5.5 ± 3.2	36
Defendant	70.0	63.6	66.7	67.6	75.2	71.1	4.4 ± 1.5	33
Agent	64.7	34.4	44.9	55.9	40.6	46.8	1.9 ± 6.8	32
Victim	65.0	56.5	60.5	69.9	82.6	75.7	15.3 ± 4.3	23
Org	68.2	68.2	68.2	67.9	78.2	72.6	4.4 ± 1.9	22
Buyer	36.8	38.9	37.8	35.9	45.5	40.1	2.2 ± 6.1	18
Target	20.0	6.3	9.5	61.4	55.0	57.8	48.3 ± 2.0	16
Seller	63.6	43.8	51.9	100.0	7.5	13.9	-38.0 ± 4.7	16
Adjudicator	57.1	26.7	36.4	42.8	48.0	45.1	8.8 ± 3.3	15
Position	72.7	61.5	66.7	84.5	100.0	91.6	25.0 ± 1.8	13
Instrument	73.3	84.6	78.6	71.0	78.4	74.5	-4.1 ± 2.1	13
Giver	63.6	58.3	60.9	68.0	63.3	65.2	4.4 ± 4.9	12
Money	68.8	91.7	78.6	87.9	83.3	85.5	6.9 ± 2.0	12
Origin	50.0	27.3	35.3	34.6	49.1	40.4	5.1 ± 2.6	11
Recipient	50.0	50.0	50.0	41.4	54.0	46.7	-3.3 ± 7.5	10
Sentence	64.3	100.0	78.3	61.8	100.0	76.4	-1.9 ± 3.0	9
Plaintiff	100.0	37.5	54.5	72.2	67.5	69.3	14.8 ± 3.8	8
Price	83.3	83.3	83.3	59.3	46.7	51.0	-32.3 ± 15.5	6
Beneficiary	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ± 0.0	5
Vehicle	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ± 0.0	1
Prosecutor	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ± 0.0	0

Table A.3: Test set precision, recall, and F_1 for the baseline and EVENTOR, ordered by argument type frequency. Reported are argument types with more than 30 instances. EVENTOR numbers are averaged over five test set runs. "**Mirco**" and "**no CNN**" report micro-averaged numbers for Experiments 1 and 2, respectively; the other rows report numbers for Experiment 3. "$\Delta F_1 \pm \sigma$" reports the difference in F_1 between EVENTOR and the baseline, as well as the respective standard deviation. "Support" reports the respective number of instances. † means statistically significant for every training and testing round at the $p < 0.05$ level. We measured significance only for micro-averaged numbers.

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	78.9	67.0	72.5	76.5	65.0	70.3	61.3	52.0	56.3	58.0	49.1	53.2
2	78.7	65.7	71.6	76.0	63.4	69.1	60.5	51.1	55.4	57.0	48.1	52.2
3	82.1	63.6	71.7	79.5	61.6	69.4	64.7	47.5	54.8	63.2	46.4	53.5
4	80.9	64.3	71.6	78.9	62.7	69.9	61.9	51.5	56.2	58.6	48.8	53.2
5	78.9	64.5	71.0	76.7	62.7	69.0	63.1	50.1	55.9	60.0	47.6	53.1
$\bar{x}$	79.9	65.0	71.7	77.5	63.1	69.5	62.3	50.4	55.7	59.4	48.0	53.0
σ	1.5	1.3	0.5	1.6	1.3	0.6	1.6	1.8	0.6	2.4	1.1	0.5

Table A.4: Split 1, $\theta_{\emptyset;0.0}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	78.0	68.4	72.9	75.4	66.1	70.5	63.3	52.9	57.7	60.7	50.8	55.3
2	78.7	68.0	72.9	76.6	66.1	71.0	64.9	51.3	57.3	62.3	49.2	55.0
3	80.8	66.1	72.8	78.6	64.3	70.8	64.8	51.5	57.4	62.8	49.9	55.6
4	78.4	69.1	73.4	76.0	67.0	71.3	61.9	54.8	58.1	58.3	51.6	54.8
5	80.3	63.9	71.1	78.3	62.3	69.4	63.4	51.0	56.5	62.0	49.9	55.3
$\bar{x}$	79.2	67.1	72.6	77.0	65.2	70.6	63.7	52.3	57.4	61.2	50.3	55.2
σ	1.2	2.1	0.9	1.4	1.9	0.7	1.2	1.6	0.6	1.8	0.9	0.3

Table A.5: Split 1, $\theta_{\emptyset;0.5}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	70.7	71.4	71.0	67.8	68.4	68.1	59.0	55.2	57.0	56.1	52.5	54.2
2	70.6	73.2	71.9	67.8	70.2	69.0	59.9	55.0	57.4	57.7	52.9	55.2
3	72.7	70.7	71.7	70.3	68.4	69.4	61.6	52.6	56.8	58.6	50.0	53.9
4	72.0	72.0	72.0	68.9	68.9	68.9	61.8	54.3	57.8	59.3	52.1	55.5
5	71.3	70.7	71.0	68.6	68.0	68.3	60.7	52.1	56.1	58.3	50.0	53.8
$\bar{x}$	71.5	71.6	71.5	68.7	68.8	68.7	60.6	53.8	57.0	58.0	51.5	54.5
σ	0.9	1.0	0.5	1.0	0.9	0.5	1.2	1.4	0.6	1.2	1.4	0.8

Table A.6: Split 1, $\theta_{\emptyset;0.9}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	80.7	65.5	72.3	78.4	63.6	70.3	60.9	54.5	57.5	58.5	52.4	55.3
2	79.3	67.7	73.0	76.9	65.7	70.8	60.6	56.8	58.6	58.0	54.4	56.1
3	78.9	67.0	72.5	76.5	65.0	70.3	60.1	52.7	56.2	57.2	50.2	53.5
4	82.3	62.5	71.1	79.9	60.7	69.0	63.7	49.3	55.6	61.1	47.4	53.4
5	81.1	65.5	72.5	78.9	63.6	70.4	62.6	52.3	57.0	59.5	49.7	54.1
$\bar{x}$	80.5	65.6	72.3	78.1	63.7	70.2	61.6	53.1	57.0	58.9	50.8	54.5
σ	1.4	2.0	0.7	1.4	1.9	0.7	1.5	2.8	1.2	1.5	2.7	1.2

Table A.7: Split 1, $\theta_{\text{GCN};0.0}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	79.3	66.1	72.1	77.1	64.3	70.1	60.7	52.8	56.5	58.9	51.3	54.8
2	78.5	66.4	71.9	75.8	64.1	69.5	61.3	52.2	56.4	58.3	49.7	53.7
3	79.5	68.0	73.3	76.6	65.5	70.6	62.0	54.0	57.7	60.3	52.6	56.2
4	77.7	68.2	72.6	75.1	65.9	70.2	62.5	54.1	58.0	59.9	52.0	55.7
5	78.5	65.5	71.4	76.0	63.4	69.1	62.5	51.7	56.6	59.5	49.2	53.9
$\bar{x}$	78.7	66.8	72.3	76.1	64.6	69.9	61.8	53.0	57.0	59.4	51.0	54.9
σ	0.7	1.2	0.7	0.8	1.0	0.6	0.8	1.1	0.8	0.8	1.5	1.1

Table A.8: Split 1, $\theta_{\text{GCN};0.5}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	70.5	73.0	71.7	67.7	70.0	68.8	60.2	56.3	58.2	58.0	54.3	56.1
2	72.4	72.7	72.6	69.7	70.0	69.8	60.0	58.8	59.4	57.6	56.6	57.1
3	69.9	73.4	71.6	67.3	70.7	69.0	57.5	57.9	57.7	54.9	55.2	55.1
4	70.9	72.0	71.5	68.0	69.1	68.5	59.7	53.9	56.7	58.0	52.4	55.0
5	71.8	71.8	71.8	69.8	69.8	69.8	62.3	54.8	58.3	59.7	52.5	55.9
$\bar{x}$	71.1	72.6	71.8	68.5	69.9	69.2	59.9	56.3	58.1	57.6	54.2	55.8
σ	1.0	0.7	0.4	1.2	0.6	0.6	1.7	2.1	1.0	1.7	1.8	0.9

Table A.9: Split 1, $\theta_{\text{GCN};0.9}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	80.0	64.5	71.4	77.7	62.7	69.4	61.1	52.6	56.5	58.4	50.3	54.1
2	80.5	63.6	71.1	78.2	61.8	69.0	59.9	52.4	55.9	57.4	50.2	53.6
3	81.4	62.7	70.9	80.2	61.8	69.8	61.1	50.4	55.3	58.7	48.5	53.1
4	80.7	61.8	70.0	78.3	60.0	68.0	62.2	48.3	54.3	60.3	46.8	52.7
5	79.5	62.7	70.1	78.1	61.6	68.9	60.4	50.3	54.9	57.8	48.1	52.5
$\bar{x}$	80.4	63.1	70.7	78.5	61.6	69.0	60.9	50.8	55.4	58.5	48.8	53.2
σ	0.7	1.0	0.6	1.0	1.0	0.7	0.9	1.8	0.9	1.1	1.5	0.7

Table A.10: Split 1, $\theta_{\text{treeLSTM};0.0}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	78.1	66.4	71.7	75.7	64.3	69.5	61.3	52.3	56.5	59.3	50.5	54.6
2	78.0	65.9	71.4	76.1	64.3	69.7	61.9	53.7	57.5	58.9	51.1	54.7
3	80.2	65.5	72.1	78.6	64.1	70.6	62.5	55.1	58.6	60.5	53.4	56.7
4	78.7	65.7	71.6	76.3	63.6	69.4	61.5	50.7	55.6	59.7	49.1	53.9
5	80.1	66.6	72.7	77.6	64.5	70.5	62.4	53.2	57.4	59.3	50.5	54.6
$\bar{x}$	79.0	66.0	71.9	76.9	64.2	69.9	61.9	53.0	57.1	59.5	50.9	54.9
σ	1.1	0.5	0.5	1.2	0.3	0.6	0.5	1.6	1.1	0.6	1.6	1.1

Table A.11: Split 1, $\theta_{\text{treeLSTM};0.5}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	70.0	73.6	71.8	67.4	70.9	69.1	57.8	56.6	57.2	54.8	53.6	54.2
2	71.6	71.1	71.4	69.3	68.9	69.1	58.8	55.2	57.0	56.6	53.2	54.8
3	72.7	71.4	72.0	70.4	69.1	69.7	62.8	54.7	58.5	60.4	52.6	56.2
4	71.1	71.6	71.3	68.4	68.9	68.6	60.0	53.6	56.6	57.5	51.4	54.3
5	73.1	73.0	73.0	70.4	70.2	70.3	60.3	55.8	57.9	57.5	53.3	55.3
$\bar{x}$	71.7	72.1	71.9	69.2	69.6	69.4	59.9	55.2	57.4	57.4	52.8	55.0
σ	1.2	1.1	0.7	1.3	0.9	0.7	1.9	1.1	0.8	2.0	0.9	0.8

Table A.12: Split 1, $\theta_{\text{treeLSTM};0.9}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	79.5	62.6	70.1	75.2	59.2	66.2	65.5	48.5	55.7	60.7	45.0	51.7
2	75.5	66.2	70.6	71.4	62.6	66.7	63.2	52.0	57.1	58.8	48.4	53.1
3	72.3	69.1	70.7	68.2	65.2	66.6	59.6	53.1	56.1	55.7	49.6	52.4
4	75.8	65.2	70.1	71.8	61.8	66.4	62.6	52.1	56.9	58.0	48.3	52.7
5	75.9	66.2	70.7	72.1	62.9	67.2	60.7	52.6	56.4	56.3	48.9	52.3
$\bar{x}$	75.8	65.9	70.4	71.7	62.3	66.6	62.3	51.7	56.4	57.9	48.0	52.4
σ	2.6	2.3	0.3	2.5	2.2	0.4	2.3	1.8	0.6	2.0	1.8	0.5

Table A.13: Split 2, $\theta_{\emptyset;0.0}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	75.4	68.1	71.5	71.4	64.4	67.7	62.1	52.8	57.1	58.7	49.8	53.9
2	75.7	68.7	72.0	71.4	64.7	67.9	63.1	52.0	57.0	58.8	48.4	53.1
3	76.5	66.2	71.0	72.7	62.9	67.5	63.1	52.0	57.0	58.8	48.4	53.1
4	76.1	67.7	71.7	72.6	64.6	68.4	65.6	50.7	57.2	61.0	47.2	53.2
5	77.2	66.1	71.2	72.9	62.5	67.3	63.5	48.9	55.2	59.3	45.7	51.6
$\bar{x}$	76.2	67.4	71.5	72.2	63.8	67.8	63.5	51.3	56.7	59.3	47.9	53.0
σ	0.7	1.2	0.4	0.7	1.0	0.4	1.3	1.5	0.8	1.0	1.5	0.8

Table A.14: Split 2, $\theta_{\emptyset;0.5}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	66.9	74.5	70.5	63.0	70.1	66.3	59.2	56.1	57.6	55.7	52.8	54.2
2	68.9	74.5	71.6	65.1	70.3	67.6	59.5	56.7	58.0	55.4	52.8	54.0
3	66.0	76.4	70.8	62.3	72.1	66.9	58.1	59.1	58.6	53.5	54.4	53.9
4	66.6	76.6	71.2	62.8	72.2	67.2	58.1	60.2	59.1	54.3	56.2	55.3
5	67.3	75.1	71.0	63.0	70.3	66.5	57.3	57.9	57.6	53.2	53.7	53.4
$\bar{x}$	67.1	75.4	71.0	63.2	71.0	66.9	58.4	58.0	58.2	54.4	54.0	54.2
σ	1.1	1.0	0.4	1.1	1.1	0.5	0.9	1.7	0.7	1.1	1.4	0.7

Table A.15: Split 2, $\theta_{\emptyset;0.9}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	74.9	66.0	70.2	70.8	62.4	66.3	61.8	52.6	56.9	57.9	49.3	53.2
2	76.4	67.4	71.6	72.2	63.7	67.7	63.2	55.1	58.9	59.2	51.6	55.1
3	74.9	67.6	71.1	70.9	64.0	67.3	62.2	52.4	56.9	57.9	48.7	52.9
4	74.5	67.2	70.7	70.6	63.7	67.0	63.0	51.3	56.5	58.7	47.8	52.7
5	74.2	68.6	71.3	69.8	64.5	67.1	59.4	55.3	57.2	55.0	51.3	53.1
$\bar{x}$	75.0	67.4	71.0	70.9	63.7	67.1	61.9	53.3	57.3	57.7	49.7	53.4
σ	0.8	0.9	0.5	0.9	0.8	0.5	1.5	1.8	0.9	1.6	1.7	1.0

Table A.16: Split 2, $\theta_{\text{GCN};0.0}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	72.7	70.3	71.5	68.5	66.2	67.4	59.7	57.0	58.3	55.0	52.6	53.8
2	73.4	69.3	71.3	69.0	65.2	67.0	60.6	53.6	56.9	56.5	50.0	53.0
3	76.5	69.4	72.8	72.8	66.1	69.3	62.8	54.3	58.2	59.0	51.0	54.7
4	74.5	69.2	71.8	70.5	65.5	67.9	63.7	53.4	58.1	59.0	49.5	53.8
5	76.6	66.1	71.0	72.6	62.7	67.3	62.0	53.5	57.4	57.6	49.6	53.3
$\bar{x}$	74.7	68.9	71.7	70.7	65.1	67.8	61.8	54.4	57.8	57.4	50.5	53.7
σ	1.8	1.6	0.7	2.0	1.4	0.9	1.6	1.5	0.6	1.7	1.3	0.6

Table A.17: Split 2, $\theta_{\text{GCN};0.5}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	65.1	76.8	70.5	61.0	72.0	66.0	55.8	61.7	58.6	51.8	57.3	54.4
2	68.5	74.8	71.5	64.5	70.4	67.3	58.8	60.0	59.4	55.2	56.3	55.7
3	67.1	75.4	71.0	63.6	71.5	67.3	57.9	59.0	58.4	53.9	55.0	54.4
4	66.2	75.2	70.4	62.4	70.8	66.3	58.0	59.3	58.6	54.4	55.6	55.0
5	67.5	75.9	71.4	63.4	71.3	67.1	58.0	58.4	58.2	53.8	54.1	53.9
$\bar{x}$	66.9	75.6	71.0	63.0	71.2	66.8	57.7	59.7	58.6	53.8	55.7	54.7
σ	1.3	0.8	0.5	1.3	0.6	0.6	1.1	1.3	0.5	1.3	1.2	0.7

Table A.18: Split 2, $\theta_{\text{GCN};0.9}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	76.9	65.0	70.4	72.8	61.5	66.7	60.6	53.4	56.8	56.1	49.5	52.6
2	76.7	64.1	69.8	72.8	60.8	66.2	61.8	52.0	56.5	58.3	49.0	53.3
3	75.2	65.1	69.8	71.2	61.6	66.1	60.9	52.0	56.1	57.2	48.9	52.7
4	76.4	65.9	70.8	72.3	62.4	67.0	61.5	51.1	55.8	57.1	47.5	51.8
5	74.5	68.5	71.4	70.5	64.7	67.5	59.8	57.2	58.5	55.0	52.6	53.8
$\bar{x}$	75.9	65.7	70.4	71.9	62.2	66.7	60.9	53.1	56.7	56.7	49.5	52.8
σ	1.0	1.7	0.7	1.0	1.5	0.6	0.8	2.4	1.1	1.2	1.9	0.8

Table A.19: Split 2, $\theta_{\text{treeLSTM};0.0}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	76.1	64.6	69.9	72.3	61.4	66.4	62.6	51.1	56.3	58.3	47.6	52.4
2	77.2	65.4	70.8	73.1	62.0	67.1	63.5	50.4	56.2	59.4	47.1	52.5
3	76.6	67.4	71.7	72.6	63.9	68.0	63.2	51.1	56.5	59.5	48.1	53.2
4	76.8	66.5	71.2	72.7	62.9	67.5	61.1	53.7	57.2	56.9	50.1	53.3
5	74.7	69.0	71.7	70.3	65.0	67.5	61.8	52.9	57.0	57.4	49.2	53.0
$\bar{x}$	76.3	66.6	71.1	72.2	63.0	67.3	62.4	51.8	56.6	58.3	48.4	52.9
σ	1.0	1.7	0.8	1.1	1.4	0.6	1.0	1.4	0.4	1.2	1.2	0.4

Table A.20: Split 2, $\theta_{\text{treeLSTM};0.5}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	68,4	73,9	71,1	64,5	69,7	67,0	58,5	58,4	58,4	54,6	54,6	54,6
2	70,9	72,5	71,7	67,0	68,6	67,8	59,3	57,5	58,4	55,3	53,6	54,5
3	68,4	75,5	71,8	64,6	71,4	67,8	60,3	58,7	59,5	56,8	55,2	56,0
4	68,1	74,4	71,1	64,1	70,0	66,9	58,3	58,1	58,2	55,1	54,9	55,0
5	68,9	73,7	71,2	65,1	69,7	67,3	59,7	57,6	58,6	55,6	53,7	54,7
$\bar{x}$	68.9	74.0	71.4	65.1	69.9	67.4	59.2	58.1	58.6	55.5	54.4	55.0
σ	1.1	1.1	0.3	1.1	1.0	0.4	0.8	0.5	0.5	0.8	0.7	0.6

Table A.21: Split 2, $\theta_{\text{treeLSTM};0.9}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	78.9	66.5	72.2	75.4	63.5	68.9	70.6	53.5	60.9	66.7	50.5	57.5
2	77.4	66.4	71.5	73.8	63.3	68.2	69.0	55.2	61.3	65.0	52.1	57.8
3	79.4	65.1	71.5	75.7	62.1	68.2	70.5	51.8	59.7	65.8	48.3	55.8
4	76.1	67.9	71.8	72.6	64.8	68.5	69.7	54.6	61.2	64.4	50.4	56.5
5	78.3	66.7	72.0	74.6	63.6	68.7	69.8	55.4	61.8	65.4	51.9	57.8
average	78.0	66.5	71.8	74.4	63.5	68.5	69.9	54.1	61.0	65.5	50.6	57.1
stdev	1.3	1.0	0.3	1.3	1.0	0.3	0.7	1.5	0.8	0.9	1.5	0.9

Table A.22: Split 3, $\theta_{\emptyset;0.0}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	77.0	68.0	72.2	73.5	64.9	68.9	68.5	57.1	62.3	64.4	53.7	58.5
2	79.0	67.2	72.6	75.6	64.3	69.5	72.6	52.4	60.9	67.6	48.8	56.7
3	78.5	66.3	71.9	74.9	63.2	68.6	68.9	55.0	61.1	63.9	51.0	56.7
4	77.5	66.8	71.8	74.0	63.8	68.5	69.5	54.7	61.2	64.6	50.9	57.0
5	77.5	67.3	72.1	74.1	64.3	68.8	69.9	55.3	61.7	66.1	52.3	58.4
average	77.9	67.1	72.1	74.4	64.1	68.9	69.9	54.9	61.4	65.3	51.3	57.5
stdev	0.8	0.6	0.3	0.8	0.6	0.4	1.6	1.7	0.6	1.5	1.8	0.9

Table A.23: Split 3, $\theta_{\emptyset;0.5}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	71.1	74.0	72.5	67.9	70.7	69.3	68.1	60.2	63.9	63.8	56.4	59.8
2	69.7	76.8	73.1	66.6	73.4	69.9	68.0	60.1	63.8	63.7	56.2	59.7
3	70.8	74.1	72.4	67.7	70.9	69.3	68.4	59.6	63.7	64.7	56.4	60.2
4	70.6	75.4	72.9	67.6	72.2	69.8	67.3	63.0	65.1	62.8	58.9	60.8
5	69.2	76.4	72.6	65.9	72.8	69.2	66.4	62.5	64.4	62.1	58.4	60.2
average	70.3	75.3	72.7	67.1	72.0	69.5	67.6	61.1	64.2	63.4	57.3	60.1
stdev	0.8	1.3	0.3	0.9	1.2	0.3	0.8	1.6	0.6	1.0	1.3	0.4

Table A.24: Split 3, $\theta_{\emptyset;0.9}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	77.3	67.0	71.8	73.7	63.9	68.5	67.9	57.3	62.1	63.8	53.8	58.4
2	77.2	65.8	71.0	73.5	62.6	67.6	69.1	56.1	61.9	64.9	52.7	58.2
3	78.0	66.8	72.0	74.5	63.8	68.8	68.3	57.9	62.7	63.4	53.7	58.2
4	77.5	66.8	71.8	73.9	63.7	68.4	69.8	56.0	62.1	65.0	52.2	57.9
5	76.9	66.2	71.2	73.4	63.2	68.0	69.1	53.4	60.2	65.0	50.2	56.6
average	77.4	66.5	71.6	73.8	63.4	68.3	68.8	56.1	61.8	64.4	52.5	57.9
stdev	0.4	0.5	0.4	0.4	0.5	0.5	0.7	1.7	0.9	0.8	1.5	0.7
	-0.6	0.0	-0.2	-0.6	0.0	-0.2	-1.1	2.0	0.8	-1.0	1.9	0.8

Table A.25: Split 3, $\theta_{\text{GCN};0.0}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	76.9	68.0	72.2	73.5	65.0	69.0	68.8	57.1	62.4	64.4	53.4	58.4
2	77.0	68.8	72.6	73.6	65.8	69.5	70.4	57.3	63.2	66.0	53.8	59.3
3	77.8	68.2	72.7	74.4	65.3	69.6	69.0	58.0	63.1	64.5	54.2	58.9
4	77.1	68.7	72.7	73.9	65.8	69.6	69.7	57.3	62.9	65.5	53.9	59.1
5	77.2	68.9	72.8	73.6	65.7	69.4	70.2	54.9	61.6	66.0	51.6	57.9
average	77.2	68.5	72.6	73.8	65.5	69.4	69.6	56.9	62.6	65.3	53.4	58.7
stdev	0.4	0.4	0.2	0.4	0.4	0.2	0.7	1.2	0.7	0.8	1.0	0.6
	-0.7	1.4	0.5	-0.6	1.4	0.6	-0.3	2.0	1.2	0.0	2.0	1.3

Table A.26: Split 3, $\theta_{\text{GCN};0.5}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	70.2	74.8	72.4	67.2	71.7	69.4	66.3	63.0	64.6	62.1	59.0	60.5
2	69.4	75.7	72.4	66.5	72.6	69.4	67.2	63.1	65.1	63.2	59.4	61.2
3	69.4	76.6	72.8	65.9	72.8	69.2	65.7	63.6	64.6	61.1	59.1	60.1
4	70.2	76.0	73.0	66.9	72.5	69.6	65.7	63.3	64.5	61.3	59.1	60.1
5	71.5	75.7	73.5	68.3	72.3	70.2	66.5	63.4	64.9	62.2	59.2	60.7
average	70.1	75.8	72.8	67.0	72.4	69.6	66.3	63.3	64.7	62.0	59.2	60.5
stdev	0.9	0.7	0.5	0.9	0.4	0.4	0.6	0.2	0.3	0.8	0.2	0.5
	-0.1	0.4	0.1	-0.2	0.4	0.1	-1.4	2.2	0.6	-1.4	1.9	0.4

Table A.27: Split 3, $\theta_{\text{GCN};0.9}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	76.7	66.8	71.4	73.3	63.8	68.2	70.1	53.2	60.5	66.1	50.2	57.0
2	76.6	67.5	71.8	73.2	64.5	68.6	69.0	58.2	63.1	64.7	54.6	59.2
3	75.6	68.9	72.1	72.2	65.8	68.8	68.0	59.2	63.3	63.3	55.1	58.9
4	77.8	66.9	71.9	74.3	63.9	68.7	68.5	57.1	62.3	64.4	53.6	58.5
5	77.2	68.4	72.5	73.6	65.2	69.1	70.2	56.3	62.5	65.5	52.6	58.3
average	76.8	67.7	71.9	73.3	64.6	68.7	69.2	56.8	62.3	64.8	53.2	58.4
stdev	0.8	0.9	0.4	0.8	0.9	0.3	1.0	2.3	1.1	1.1	1.9	0.8

Table A.28: Split 3, $\theta_{\text{treeLSTM};0.0}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	75.0	68.9	71.8	71.4	65.6	68.4	67.1	60.5	63.6	63.2	57.0	60.0
2	77.1	69.4	73.0	73.8	66.3	69.9	68.8	59.5	63.8	64.5	55.9	59.9
3	77.7	68.6	72.8	74.1	65.4	69.4	68.6	58.1	62.9	64.6	54.7	59.3
4	76.5	69.0	72.5	72.8	65.7	69.0	68.8	57.5	62.7	64.3	53.7	58.5
5	76.8	69.9	73.2	73.5	66.9	70.1	70.2	58.0	63.5	64.3	53.1	58.2
average	76.6	69.2	72.7	73.1	66.0	69.4	68.7	58.7	63.3	64.2	54.9	59.2
stdev	1.0	0.5	0.5	1.1	0.6	0.7	1.1	1.2	0.5	0.6	1.6	0.8

Table A.29: Split 3, $\theta_{\text{treeLSTM};0.5}$

	Trigger Identification			Trigger Classification			Argument Identification			Argument Classification		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
1	70.4	74.7	72.5	67.2	71.3	69.2	66.1	62.6	64.3	62.0	58.8	60.4
2	70.7	76.1	73.3	67.5	72.7	70.0	68.4	60.4	64.1	64.1	56.6	60.1
3	70.4	75.8	73.0	67.2	72.5	69.7	67.1	61.7	64.3	62.7	57.6	60.1
4	68.1	76.7	72.1	65.0	73.2	68.9	64.6	63.2	63.9	60.5	59.2	59.8
5	70.9	75.5	73.1	67.9	72.3	70.0	66.9	62.6	64.7	62.9	58.8	60.8
average	70.1	75.8	72.8	67.0	72.4	69.6	66.6	62.1	64.3	62.4	58.2	60.2
stdev	1.1	0.7	0.5	1.1	0.7	0.5	1.4	1.1	0.3	1.3	1.1	0.4

Table A.30: Split 3, $\theta_{\text{treeLSTM};0.9}$

Split 1	θ_{GCN}	argument F_1	0.028971028971029
		trigger F_1	0.228771228771229
	θ_{treeLSTM}	argument F_1	0.569430569430569
		trigger F_1	0.150849150849151
Split 2	θ_{GCN}	argument F_1	0.244755244755245
		trigger F_1	0.867132867132867
	θ_{treeLSTM}	argument F_1	0.223776223776224
		trigger F_1	0.081918081918082
Split 3	θ_{GCN}	argument F_1	0.521478521478521
		trigger F_1	0.839160839160839
	θ_{treeLSTM}	argument F_1	0.939060939060939
		trigger F_1	0.744255744255744

Table A.31: Statistical significance of the respective F_1 measure averaged over all five evaluation runs compared to the same $\theta_\emptyset$ measure. Lower numbers mean higher significance. We use approximate randomization as a test (Noreen, 1989)