SPECIFICATION-BASED EVENT EXTRACTION

Ofer Bronstein

Submitted in partial fulfillment of the requirements for the Master’s Degree in the Department of Computer Science, Bar-Ilan University

Ramat-Gan, Israel 2015
This work was carried out under the supervision of
Prof. Ido Dagan
Department of Computer Science, Bar-Ilan University.
Acknowledgments

This thesis was completed thanks to the help and guidance of quite a few people whom I would like to thank deeply.

First and foremost I would like to thank my advisor, Prof. Ido Dagan. It was with a great sense of admiration and awe that I first encountered him when attending my first NLP class which he taught, a sense that grew stronger as the years went by and I got to see more aspects of him as a dedicated teacher, creative researcher, motivated manager, and just generally a good guy. Ido walked me through my time as a student with much caring and enthusiasm, while maintaining an amazing community in the form of his NLP lab at Bar-Ilan, which never failed at containing nothing but excellent people. Ido is truly one of a kind, and I am very fortunate to have been his student.

Second, I would like to thank everyone from the the Bar-Ilan university NLP lab, in all of its incarnations during my period - Asher Stern, Eyal Shnarch, Naomi Zeichner, Hila Weisman, Jonathan Berant, Erel Segal Halevi, Meni Adler, Lili Kotlerman, Chaya Liebeskind, Amnon Lotan, Oren Melamud, Omer Levy, Yoav Goldberg, Micah Shlain, Gabriel Satanovsky, Jessica Ficler, Vasily Konovalov and Vered Shwartz. All are great colleagues, but even better friends.

I would like to thank Qi Li and Prof. Heng Ji from RPI University in New York, and Prof. Anette Frank from Heidelberg University in Germany, for their dedicated collaboration with me in the last two years over my research. Their useful advice and guidance contributed greatly to my work, and the paper we all published together this summer was a true joint effort that was both mind-opening and enjoyable.

I would also like to thank the various people of the NLP community from around the globe that I got to meet since I entered this exciting field. People from Israel, Germany, the United States, Italy, and many more places, who are too many to mention here. It has been an adventurous ride.

And last I would like to thank my family and friends, who, throughout the ups and down of this aforementioned crazy ride, were always loving enough to say the right word.
Contents

1 Introduction 1

2 Background 4

2.1 ACE 2005 Event Extraction Task 4

2.1.1 Annotation Guidelines 5

2.1.2 Dataset 6

2.2 Fully-Supervised Event Extraction Systems 8

2.3 A State-of-the-Art Fully-Supervised Event Extraction System 9

2.3.1 Decoding 9

2.3.2 Assignment Scoring 13

2.3.3 Feature Types 16

2.3.4 Training 18

3 Seed-Based Trigger Labeling 20

3.1 Problem Setup 20

3.1.1 Seed Lists 21

3.1.2 Features 22

3.1.3 Training 23

3.1.4 Summary 23

3.2 Method 24

3.2.1 Labels 25

3.2.2 Assignment Scoring 25

3.2.3 Feature Types 28

3.3 Evaluation 31

3.3.1 Setting 31
3.3.2 Results .......................................................... 36
3.3.3 Error Analysis ................................................. 37
3.4 Related Work ..................................................... 43

4 Specification-Based Event Extraction .......................... 44
4.1 Problem Setup .................................................. 44
4.2 Method ........................................................... 47
  4.2.1 Labels ......................................................... 47
  4.2.2 Assignment Scoring ....................................... 49
  4.2.3 Feature Types .............................................. 50
    4.2.3.1 Stand-Alone Argument Features .................. 50
    4.2.3.2 Linked Argument Features ......................... 51
4.3 Evaluation ..................................................... 54
  4.3.1 Setting ....................................................... 54
  4.3.2 Results ..................................................... 55

5 Conclusions and Future Work .................................. 57
  5.1 Conclusions .................................................. 57
  5.2 Future Work .................................................. 58
List of Figures

2.1 ACE-2005 event description of Convict, triggers and arguments. . 7
2.2 Full labeling of a sentence for triggers and arguments, including arguments that are common for both events triggered by “died” and “fired”. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.1 Flow of the seed-based approach . . . . . . . . . . . . . . . . . 24
4.1 Arguments portion of the ACE-2005 event description of Convict. 45
4.2 Spec structure . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
4.3 Event spec XML (partial). Triggers and arguments are marked in the usage samples in brackets, via identifying markers defined for each of them. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.4 Dependency graphs of sentences from text input and from the usage samples portion of the spec. Both graphs have the following dependency path marked in bold: [T] (prep(pobj[A])) . . . 52
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Local and global feature types</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Seed lists of several event types collected from Section 5 of the ACE-2005 guidelines</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Calculation of feature vector $f(x, i, y, t)$, where $x_i$=&quot;He perished at noon.&quot;, $y=(\bot, \top, \bot, \bot)$ and $t=Die$. The complete feature vector can be seen by scanning down column $f_\phi(x, i, y, t)$. It has exactly 8 slots, one for each feature instance $\phi$ in the system (2 labels ${\top, \bot}$ times 4 feature types, described later in section 3.2.3)</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>Similarity feature types implemented in the system, using only the WordNet knowledge resource</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Split of the ACE-2005 documents, used in our evaluation as well as previous evaluations: (Ji and Grishman [2008]; Liao and Grishman [2010]; Li et al. [2013])</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>20 out of 330 evaluation runs. Each run has a global identifier ($G$) and a per-test-type identifier ($T$). For each event type or list of event types we specify the amount of annotated triggers we use ($tr$), within the corresponding documents. For the test event type we use all available annotated mentions in the test documents, whereas for the training and development event types we use fixed sampled amounts - 24 and 6 respectively. Finally, we show the $F_1$ score of each run, as a “sneak-peek” to our results. Note the low variability of $F_1$ for each test event type across different runs.</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>Seed lists manually collected for our evaluation from Section 5 of the ACE-2005 guidelines</td>
<td>35</td>
</tr>
</tbody>
</table>
3.7 Seed-Based performance compared to fully-supervised systems, plus average and variance of $F_1$ variance (%) over the 10 test runs per test event type. .................................................. 36

3.8 Distribution of reasons for false-positive errors. Most error reasons are caused by the system, yet one originates in the gold standard labeling. .......................................................... 37

3.9 Examples for false-positive Wrong candidate token synset. ........................................... 38

3.10 Examples for false-positive Wrong seed synset ................................................................. 39

3.11 Examples for false-positive Different word choice for trigger ........................................ 39

3.12 Examples for false-positive Candidate token too general .............................................. 40

3.13 Examples for false-positive Missing gold annotation ...................................................... 40

3.14 Distribution of reasons for false-negative errors. Most error reasons are caused by the system, yet one originates in the gold standard labeling. .................................................. 41

3.15 Examples for false-negative No relation connecting candidate token and seeds ................ 41

3.16 Examples for false-negative Wrong gold annotation ....................................................... 42

3.17 Examples for false-negative Coreference ................................................................. 42

4.1 Parameters constructing different dependency-path-based linked argument features .............. 53
Abstract

Event extraction is a common task in information extraction, which aims at identifying occurrences of target event types in given texts. It comprises of two parts - trigger labeling, which identifies the main term signifying the event, and argument extraction, which identifies the participants of the event and their roles. The task is typically addressed in a fully-supervised manner, which requires a large amount of annotated data for training. Two major difficulties lie within this approach: (1) It demands significant human effort in both annotating texts, and in writing quality guidelines for the annotators. (2) This effort must be fully applied for each target event type, resulting in high costs for every required event.

In this work we propose an alternative approach to alleviate these difficulties. Given rich event descriptions taken from annotation guidelines, our method’s extraction phase checks for similarities between input texts and a target event description. When similarity is found, the inspected text snippet is labeled as an occurrence of the target event. By utilizing event descriptions directly during extraction, we can skip annotating large amounts of text of the target event, eliminating much of the required effort. Additionally, as we have an event-independent model which is not trained on the target event types, our method can extract any new event type, requiring only the limited effort of writing its event description.

We implemented our approach as a lightly-supervised event extraction system, based on a structured perceptron that uses similarity measures as classification features. We evaluated the system over the ACE-2005 event extraction dataset, and compared it to a state-of-the-art fully-supervised event extraction system. Our work was divided to two steps: In the first step, the system extracted only the event triggers (without the arguments). Here it achieved 5.7% improvement in $F_1$ over the fully-supervised system, which did use the full annotated data. This is the main contribution of our work. In the second step, the system extracted both the triggers and arguments of target events. This step did not yield good enough performance, and is brought here largely as negative results, with ideas for future improvements.
Chapter 1

Introduction

Event extraction is a prominent subtask of Information Extraction, targeted at identifying event triggers and their arguments in given texts. For example, consider the following sentence:

(1) Twenty people were wounded in last Tuesday’s airport blast.

In this case, an event extraction system is expected to extract two events:

1. An event of type Injure. The word token that most clearly expresses the event is “wounded” - this will be marked as the event trigger. The other participants in the event would be “Twenty people”, functioning as the Victim, and “last Tuesday”, stating the Time. These will be marked as the event arguments, with their respective roles in the sentence.

2. An event of type Attack. It has the trigger “blast”, and two arguments - “last Tuesday” with the role Time, and “airport” with the role Place. Note that “last Tuesday” functions as an argument in both events, which makes sense, as it participates in both (“airport” may also be considered as an argument in both events).

Typically, an event extraction system would extract a pre-determined set of event types (such as Injure and Attack above). Each event type has a specific meaning, and a defined set of argument roles, which may appear all or in part in any event mention. The goal of the system would be to identify every occurrence
of events from the given types in some input set of documents, and output each trigger with its proper event type, and each of its arguments with its proper role.

Most state-of-the-art event extraction approaches \cite{ji2008crf, liao2010, hong2011, li2013} follow the standard supervised learning paradigm. For each event type, experts first write annotation guidelines, describing what constitutes a mention of the event. These descriptions typically include a verbal specification of the event with illustrating example sentences. Then, annotators follow these guidelines to label event mentions in a large dataset. Finally, a classifier is trained over the annotated triggers and arguments to extract the target events.

A significant shortcoming of the supervised paradigm lies with the major human effort it requires. This effort is put both in producing high-quality guidelines, and in dataset annotation for each new event type. Producing effective guidelines is a demanding task, requiring much knowledge, expertise and creativity. These are utilized towards writing event descriptions that strike the nearly impossible balance between comprehensive and precise. Given this rich information embedded in the guidelines, we raise in this work the following research question:

How well can we perform by leveraging only the lexical knowledge already available in quality guidelines for new event types, without requiring annotated training data for them?

To address this question, we propose a specification-based approach for the event extraction task. Given the description for a new event type, we first collect from it all examples of triggers, arguments, and illustrating sentences, into a pre-defined structure named specification (or “spec”). Then, at the extraction phase, we go through the input text and assess the similarity of its different element to different components in the spec. Elements deemed similar enough to the spec will be outputted by the system as triggers/arguments. To continue example (1), assume the event description of event type Attack contains several trigger examples (also termed seeds), such as “explosion” and “fire”. We can then detect that the token “blast” is a hyponym of “explosion” and a synonym of “fire”, and infer that “blast” is a likely Attack trigger. A similar process can be done for arguments (as elaborated in chapter 4).
In our method, such similarity indicators between the text and the spec are encoded as a small set of event-independent classification features. These features are based on lexical and syntactic matches, utilizing external resources like WordNet. Using event-independent features allows us to train the system only once, at system setup phase, requiring annotated data in a training set for just a few pre-selected event types. Then, whenever a new event type is introduced for extraction, we only need to compose a spec for it from its description, and provide it as input to the system.

Chapter 2 describes relevant background to our work - the ACE-2005 benchmark, used for evaluating our system, and a state-of-the-art fully-supervised event extraction system (Li et al., 2013), to which we compare our performance.

We start off our contribution by utilizing the fully-supervised system, developing our spec-based system based on it. We divide our work to two steps:

1. **Seed-Based Trigger Labeling** (chapter 3): We built a system that only labels triggers in text, ignoring arguments. In this case the spec takes a simple form of a seed list, using only trigger examples (seeds) from the guidelines. We evaluated the system over the ACE-2005 dataset, with the same evaluation setting as used by the fully-supervised system, which uses annotated data for its test events. Even though we do not use any annotated data for test events, but rather only their seed lists built from the ACE annotation guidelines, the seed-based system outperforms the fully-supervised one. Additionally, we review the broader line of research on avoiding or reducing annotation cost in information extraction. In particular, we note the potential utility of our approach in scenarios where manual annotation per each new event is too costly. This chapter is based on our recently published paper (Bronstein et al., 2015).

2. **Specification-Based Event Extraction** (chapter 4): Here the system extracts the full event by labeling both triggers and arguments, utilizing the full information from the ACE annotation guidelines. Unfortunately, this endeavor did not yield good enough performance, and is thus brought largely as negative results, while suggesting possible directions for improvement.
Chapter 2

Background

This chapter provides relevant background for the current work. We first describe the specific setting of the ACE 2005 benchmark, which is a common setting for performing event extraction, and is also the setting in which we operate. Next, we describe various fully-supervised event extraction systems, that operate in the same setting. We then bring in detail the description of the state-of-the-art fully-supervised event extraction system of (Li et al., 2013). This description may be the most essential for the understanding of our system, as our implementation is based on the Li et al. (2013) one, changing only several key components (such as labels and features). Finally, we describe other works that attempted at reducing or eliminating the need for manual annotation in the field of Information Extraction (IE).

2.1 ACE 2005 Event Extraction Task

The Automatic Content Extraction (ACE) 2005 Challenge incorporates several tasks in the field of Information Extraction. We focus on its event extraction task EDR (Event Detection and Recognition), which is typically used as a benchmark for event extraction systems. We also use it as a benchmark for our system’s performance.

http://projects.ldc.upenn.edu/ace
2.1.1 Annotation Guidelines

The task defines 33 event types that an event extraction system should extract. It provides an annotation guidelines document meant for annotators, which provides all necessary definitions and information required for manual annotation of text documents. The document is logically divided into 3 parts: General Instructions, Triggers and Arguments.

**General Instructions** Sections 1-4 of the guidelines define general concepts and provide general instructions for labeling. Some important points are:

- An event trigger is the word that most clearly expresses the event occurrence. For example in “John was charged with money laundering last week”, “charged” is the trigger of an event of type Charge-Indict.

- An event argument is any entity participating in the event, and any value or time-expression that is an attribute of the event. Each argument has a specific role, depending on the event type and the context of the sentence. Continuing our example, in “John was charged with money laundering last week”, “John” is an entity argument with role Defendant, “money laundering” is a value argument with role Crime, and “last week” is a time-expression argument with role Time. These will all be linked to the event mention that has “charged” as a trigger.

- An event extent is the sentence in which an event’s trigger appears. We only label the event’s arguments that appear in this sentence, meaning that an event, as a whole, is confined to a single sentence.

- A trigger can have various parts-of-speech, such as verb (“She sued Dan...”), noun (“The explosion caused...”), pronoun (with coreference - “After the attack, they found who did it...”), and adjective (“The dead members...”).

---

2 In fact, in the ACE terminology these 33 event types are called “event subtypes”, and they are distributed among 8 event type categories named “event types”. As we have no interest in the categories, we ignore them, and refer to ACE’s event subtypes as “event types”.

3 https://www.ldc.upenn.edu/sites/www.ldc.upenn.edu/files/english-events-guidelines-v5.4.3.pdf
• A trigger can have multiple words in the rare case of verb+particle, e.g. “Jane was laid off by the firm”.

• The 3 kinds of argument candidates (entities, values and time-expressions) have internal types. Entities have types such as PER (person), ORG (organization), GPE (geo-political entity) and WEA (weapon). Values have types such as CRIME, JOB-TITLE and MONEY. Time-expressions do not have internal types (when a type is required, TIME is written).

**Triggers**  Section 5 of the guidelines provides an *event description* for each event type. It starts with a verbal specification, describing what constitutes a mention of that event. It then continues with a list of illustrating example sentences with marked triggers. The descriptions span on average less than a page per type. The top part of Figure 2.1 shows an example description for event type *Convict*.

**Arguments**  Section 6 concludes the guidelines by providing the specification of arguments for each event type. For each argument, it names a role (which explains the relation between the argument and the event), a type (of the entity / value / time-expression), a description, and an example sentence (with the argument marked in brackets). The bottom part of Figure 2.1 shows an example argument specification for event type *Convict*.

### 2.1.2 Dataset

ACE-2005 provides a training set of 599 English documents, from various sources such as newswire, blogs, and phone conversations. There is a common split of these documents to *training*, *development* and *test* (Table 3.4), used by many IE systems. Each document has two parts: The first part is a text file, containing the document’s content. The second part is an XML file, containing all the annotations created over the raw text (each annotation is grounded with character offsets over the text file). For each event it contains the trigger word, and for each of the event’s arguments it contains the entire argument phrase, its role and its type.

---

4 A test set also exists, but is not publicly available, and thus it is not commonly used in research.
Section 5 - Triggers

**Convict**
A CONVICT Event occurs whenever a TRY Event ends with a successful prosecution of the DEFENDANT-ARG. In other words, a PERSON, ORGANIZATION or GPE Entity is convicted whenever that Entity has been found guilty of a CRIME. It can have a CRIME attribute filled by a string from the text. CONVICT Events will also include guilty pleas.

**Examples:**
- *Martha Breckenridge was convicted of two counts of manslaughter.*
- *Tommy, a multimillionaire with a playboy image and love of fast cars, is the first member of Suharto’s family to be convicted of graft.*
- *It found him guilty of enriching himself through a property deal with the state’s main food supply agency.*

Section 6 - Arguments

<table>
<thead>
<tr>
<th>Role</th>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defendant</td>
<td>PER</td>
<td>The convicted agent(s)</td>
<td>A Russian court <strong>convicted</strong> [Pope] Wednesday on espionage charges and sentenced him to 20 years in prison.</td>
</tr>
<tr>
<td></td>
<td>ORG</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>GPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adjudicator</td>
<td>PER</td>
<td>The judge or court</td>
<td>[A Russian court] <strong>convicted</strong> Pope Wednesday on espionage charges...</td>
</tr>
<tr>
<td></td>
<td>ORG</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>GPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crime</td>
<td>CRIME</td>
<td>The crime for which the Defendant has been convicted</td>
<td>A Russian court <strong>convicted</strong> Pope Wednesday on [espionage] charges...</td>
</tr>
<tr>
<td>Time</td>
<td>TIME</td>
<td>When the conviction takes place</td>
<td>A Russian court <strong>convicted</strong> Pope [Wednesday] on espionage charges...</td>
</tr>
<tr>
<td>Place</td>
<td>GPE</td>
<td>Where the conviction takes place</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FAC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.1: ACE-2005 event description of *Convict*, triggers and arguments.
In Baghdad, a cameraman died when an American tank fired on the Palestine Hotel.

Figure 2.2: Full labeling of a sentence for triggers and arguments, including arguments that are common for both events triggered by “died” and “fired”.

2.2 Fully-Supervised Event Extraction Systems

Event extraction systems are typically classifier-based, and often use a sequential pipeline (Ji and Grishman, 2008; Liao and Grishman, 2010; Hong et al., 2011). They first address the subtask of identifying and classifying event triggers. Once this subtask is complete, they identify and classify the arguments of those triggers.

A common drawback for the pipelined architecture is propagation of errors from the upstream to the downstream components. More specifically, mislabeling of triggers would likely cause mislabeling of arguments, even if the argument-labeling component had the potential of improving the labeling of triggers. For example, consider the following sentence:

(1) In Baghdad, a cameraman died when an American tank fired on the Palestine Hotel.

Due to its ambiguity, the trigger “fired” might be missed or misclassified as an End-Position event, and not correctly as Attack. This is despite the fact that “tank” could be recognized as a possible Instrument argument of an Attack, suggesting the correct event type.

An additional problem with the pipelined approach is its inability to handle dependencies between multiple triggers and arguments in a sentence. Returning to our example, Figure 2.2 depicts the full labeling of sentence (1). We can see that “cameraman” is both a Victim of the Die event, and a Target of the Attack event. A pipelined approach might miss the latter connection, as it won’t be able to leverage the relation between both events in the sentence. Both drawbacks motivate for jointly classifying triggers and arguments, as described next.
2.3 A State-of-the-Art Fully-Supervised Event Extraction System

We now describe the state-of-the-art fully-supervised event extraction system of Li et al. (2013). We bring only details relevant for the understanding of our system and evaluation, as they are both heavily based on their research. Please refer to the published paper (Li et al., 2013) for full details.

Li et al. (2013) presented an event extraction system, which identifies and classifies both triggers and arguments in a joint manner, following a structured prediction approach. In the previously mentioned sentence (1), the system is able to utilize the insight that “tank” is a likely Instrument used for Attack, and thus contribute to “fired” being labeled as an Attack. Furthermore, it will be able to identify the relation between the Die and Attack events, thus correctly labeling “cameraman” as an argument in both. Indeed, this system’s evaluation shows an improvement it gains over the pipelined systems.

The input for the system is a set of text sentences, each sentence marked with all of the argument candidates it contains (all mentions of entities, time-expressions and values). The system does not attempt at identifying these argument candidates itself, and instead receives them either from the gold standard of the ACE-2005 dataset, or from an external IE system built for extracting them.

We first explain how the system, using a trained model \( w \) (a weight vector), decodes a single sentence by extracting all the event mentions in it, using beam search. Then, we explain how the system learns \( w \) during the training process, using a structured perceptron.

2.3.1 Decoding

The input for the decoding process is a sentence instance \( x \in X \), such that:

\[
x = \langle (x_1, x_2, \ldots, x_s), E \rangle
\]

where \( x_i \) represents the \( i \)-th token in the sentence (out of a total of \( s \) tokens), and \( E = \{ e_k \}_{k=1}^m \) is \( x \)'s set of \( m \) available argument candidates (sometimes abbrevi-
ated to \textit{argcands}). The output of decoding is a sentence assignment \( y \in \mathcal{Y}(x) \), such that:

\[
y = (y_1, y_{1,1}, \ldots, y_{1,m}, \ldots, y_s, y_{s,1}, \ldots, y_{s,m})
\]

where \( y_i \) represents the assigned trigger label for token \( x_i \), and \( y_{i,k} \) represents the argument role label for the edge between a trigger \( x_i \) and argument candidate \( e_k \). For example, for a sentence with \( s = 3 \) tokens and \( m = 2 \) argument candidates, our notation for assignment \( y \) would be:

\[
y = (\underbrace{\bot}_{\text{trigger label for } x_1}, \underbrace{y_{1,1}, y_{1,2}, y_2}_{\text{arg labels for args of } x_2})
\]

Trigger labels \( y_i \) are selected from the alphabet \( \mathcal{L} \cup \{\bot\} \), where \( \mathcal{L} \) represents the 33 ACE-2005 event types, and \( \bot \) (read \textit{bottom}) indicates that the token is not a trigger at all. Similarly, argument labels \( y_{i,k} \) are selected from alphabet \( \mathcal{R} \cup \{\bot\} \), where \( \mathcal{R} \) is the set of ACE-2005 argument roles, and \( \bot \) indicates that the argument candidate is not an argument for the associated trigger. As a concrete example, the variables for the toy sentence "Jobs founded Apple" are as follows:

\[
x = \left\langle \begin{array}{c} x_2 \\ \text{(Jobs, founded, Apple), } \{\text{Jobs}_{\text{PER}}, \text{Apple}_{\text{ORG}}\} \end{array} \right\rangle
\]

\[
y = (\bot, \bot, \bot, \underbrace{\text{Start\_Org}, \text{Agent, Org}}_{\text{args for founded}}, \bot, \bot, \bot)
\]

Since both trigger- and argument-alphabets consist of multiple labels, the system constitutes a multi-class classifier.

The decoding algorithm is illustrated in Algorithm 1. Its goal is finding the best assignment \( y \) for sentence instance \( x \), by choosing the highest scoring one:

\[
y = \arg\max_{y' \in \mathcal{Y}(x)} \text{score}(x, y')
\]

where \( \text{score}(x, y') \) is described in section 2.3.2. To achieve that, the algorithm maintains a beam, which is essentially a list with a fixed capacity, that at all times keeps the highest scoring \( K \) assignments \( y' \) of the current sentence \( x \). This
Input:
- $x$: sentence instance $\langle(x_1, x_2, \ldots, x_s), \mathcal{E}\rangle$
- $\hat{y}$: gold label assignment of $x$ (only when training)
- $K$: beam size
- $\mathcal{L} \cup \{\bot\}$: trigger label alphabet
- $\mathcal{R} \cup \{\bot\}$: argument label alphabet

Local Variables:
- $B$: beam of assignments ($B[0]$ is highest-scoring one)
- $buf$: temporarily holds all possible assignments in given point

Output:
- 1-best prediction $\hat{y}$ for $x$

Notation:
- $y \circ^i l$: assigning label $l$ to token $x_i$ in $y$
- $y \circ^{i,k} r$: assigning role $r$ to the edge between token $x_i$ and argcand $e_k$ in $y$

Algorithm 1: Decoding a single sentence using beam search. During test, lines 7-8 and 18-19 are omitted.
method, which uses beam search in a greedy fashion, is required since the search space is too large to be fully explored.

We note that during the algorithm’s run, assignment $y'$ would be a partial assignment, that contains labels only for the tokens, as well as their argument candidates, that were processed until that point. A partial assignment is notated as $y'_{[i,k]}$, such that:

$$y'_{[i,k]} = (y_1, y_{1,1}, \ldots, y_{1,m}, y_2, y_{2,1}, \ldots, y_{2,m}, \ldots, y_i, y_{i,1}, \ldots, y_{i,k})$$

where $i$ is the last processed token position and $k$ is its last processed argument candidate position. According to this definition, $y'_{[i,k]}$ contains all trigger labels up to $i$, and all $m$ argument labels for tokens before $i$, where token $i$ itself has argument labels only for the first $k$ argument candidates ($k \leq m$). Continuing our example of a sentence with $s = 3$ tokens and $m = 2$ argument candidates, partial assignment $y'_{[2,1]}$ would be:

$$y'_{[2,1]} = (y_1, y_{1,1}, y_{1,2}, y_2, y_{2,1})$$

As assignments “grow” with the progression of the algorithm, full assignments $y'[s, m]$ are created only near the end of the algorithm’s run.

We shall now describe Algorithm 1 step by step. We start with a beam containing a single empty assignment (line 1). To extract triggers, we scan the tokens of the sentence (line 2). For each token $x_i$, we go through each partial assignment currently in the beam $y'_{[i-1,0]}$ ($i - 1$ since all previous tokens already have trigger labels but not yet for $x_i$, and 0 since no argument candidate $e_k$ has any role label for $x_i$). From each partial assignment $y'_{[i-1,0]}$ we create $|L| + 1$ new assignments by assigning each given trigger label $l \in \mathcal{L}$ or $\perp$ to $x_i$ (line 3). Next, we calculate the score of each new assignment as $w \cdot f(x, y'_{[i,0]})$ (lines 4-5), the calculation is detailed later in section 2.3.2. Finally, we put in the beam only the top $K$ scoring assignments (line 6).

To extract $x_i$’s arguments, we perform a similar process in an internal loop over the current $i$. For each argument candidate $e_k \in \mathcal{E}$ (line 9), we go through each partial assignment currently in the beam $y'_{[i,k-1]}$ (now $x_i$ already has a label,
as well as all previous $k - 1$ argument candidates, but not yet for $e_k$) (line 11). From each partial assignment $\hat{y}_{i,k-1}^t$ we create $|R| + 1$ new assignments by assigning each role label $r \in R \cup \perp$ to $e_k$, or just $\perp$ if $x_i$ is not a trigger according to $\hat{y}_{i,k-1}^t$ (lines 12-14). We similarly calculate the scores of the new assignments, and put the top $K$ in the beam (lines 15-17). The decoding concludes by returning the highest-scoring assignment in the beam (line 20). The complexity of the algorithm is $O(K \cdot s \cdot m)$.

Decoding occurs in two kinds of situations: during training, when the system decodes training sentences in order to learn $w$, and during the extraction phase, operating on new sentences. During training (detailed in section 2.3.4), the algorithm consists of another component, following the approach of early-update (Collins and Roark, 2004; Huang et al, 2012): whenever partial assignments are put in the beam, we check if the relevant portion of the gold label assignment ($\hat{y}_{i,0}$ or $\hat{y}_{i,k}$) equals one of the partial assignments (lines 7-8 and 18-19). If such an assignment is found in the beam, this means that the gold assignment may eventually be found, and the algorithm continues. If not, then we are in a “violation”, and the algorithm immediately returns the top scoring assignment in the beam and finishes. This is then used for updating the model $w$ (described in section 2.3.4).

2.3.2 Assignment Scoring

Each assignment is scored according to the typical linear classification paradigm for structured prediction:

$$score(x, y) = w \cdot f(x, y) = \sum_{i=1}^{s} \sum_{k=0}^{m} w \cdot f(x, y, i, k)$$

where $s$ is the number of tokens in sentence $x$; $m$ is the number of argument candidates in $E$; $f(x, y, i, k)$ is the feature vector of assignment $y_{i,k}$ over token $x_i$ and argument candidate $e_k$; and $w$ is the weight vector that is learned for each feature instance during training.

A word on notation - many feature concepts in the system (such as feature vectors and feature instances, described later), have one of two forms: they either apply solely to triggers, and then we can discuss the token $x_i$ as a trigger candidate
and its assignment $y_i$, or they can apply to a combination of a trigger and an argument, and then we can discuss token $x_i$ with argument candidate $e_k$, and their assignment $y_{i,k}$. In the trigger-only case, we declare that $k$ gets the special value 0 (meaning no argument candidate at all), and in that case, we may omit $k$ altogether from the notation for simplicity. For example, a feature vector of token $x_i$ will be written as $f(x, y, i)$ instead of $f(x, y, i, 0)$.

We now describe how $score(x, y)$ is calculated using feature instances. The system pre-specifies a set of feature types $\mathcal{F}$ (detailed later in section 2.3.3). When a feature type $f \in \mathcal{F}$ is applied to a specific $x_i$ and $e_k$ in a sentence $x$, it records some concrete value (typically a text string or an integer), marked $f(x, i, k)$. As an example, consider the trigger-only feature type $f_1=\text{TriggerBackBigram}$, which records a trigger candidate token, and the token preceding it. When applying $f_1$ to token $i = 2$ in the sentence $x=\text{“He perished at noon.”}$, we get $f_1(x, i) = \text{ (“He”, “perished”) \ldots}$ Complementing with an example that also involves arguments, consider feature type $f_2=\text{TriggerArgLexicalDistance}$, which records the number of tokens in the sentence between the trigger and the argument candidate. When applying $f_2$ to trigger $x_2=\text{“perished”}$ and argument candidate $e_2=\text{“noon”}$ in the same sentence, we get $f_2(x, i, k) = 1$ (as the single word “at” is located between “perished” and “noon”).

When feature type $f$ is applied to some specific $x, i, k$ and $y$, we get a feature instance $\phi$ (and the creating feature type $f$ is marked as $\phi_f$). It is defined as the tuple: $\phi = (\phi_f, f(x, i, k), y_{i}, y_{i,k})$. Continuing our first example, when applying $f_1$ to $x$ with $i=2$, and assignment $y=(\perp, \text{Die}, \perp, \perp)$ (considering only the trigger portions of an assignment), we get the following feature instance $\phi_1$:

\[
\phi_1 = \phi_{f_1}(x, y, i) = \text{TriggerBackBigram(“He perished at noon.”, (\perp, \text{Die}, \perp, \perp), 2)} = \\
\langle \text{TriggerBackBigram, (“He”, “perished”), Die} \rangle
\]

This feature instance records a scenario where a token that is labeled as a Die event has a back-bigram of “He perished”. The system will learn a weight for this feature instance, which is likely to be high (as described later in this section).

Continuing with an argument example, when applying $f_2$ to $x$ with $i=2$ and
k=2, and an assignment y where $y_2=\text{Die}$ and $y_{2,2}=\text{Time}$, we get feature instance $\phi_2$:

$$
\phi_2 = \phi_f(x, y, i, k) = 
\text{TriggerArgLexicalDistance}(x, (\ldots, y_2 = \text{Die}, \ldots, y_{2,2} = \text{Time}, \ldots), 2, 2) = 
\langle \text{TriggerArgLexicalDistance}, 1, \text{Die}, \text{Time} \rangle
$$

Here the feature instance records a scenario where the lexical distance between a Die trigger and its Time argument is 1.

Note that for simplicity in the case of arguments, we bring here a definition and an example for a feature instance only of local features and not global features (defined later in section 2.3.3).

Having distinct feature instances for different trigger and argument labels allows the system to learn feature weights that are label-specific. For example, the feature instance $\langle \text{TriggerBackBigram}, (\text{“He”, “perished”}), \text{Die} \rangle$ should get a relatively high weight, due to the correlation between the text phrase “He perished” and the Die event, whereas $\langle \text{TriggerBackBigram}, (\text{“He”, “perished”}), \text{Nominate} \rangle$ should get a much lower weight, as this text phrase does not indicate a Nominate event.

The set of all feature instances $\Phi$ is constructed during the training phase (described later in section 2.3.4), using $\mathcal{F}$ and the training documents. Each $\phi \in \Phi$ is assigned a feature value slot marked $f_\phi$ and a weight slot marked $w_\phi$. While the system contains only a handful of feature types $f \in \mathcal{F}$, the highly lexicalized and label-dependent feature instances $\phi \in \Phi$ yield a very large feature space. Specifically, the system has 65 feature types, but over 360,000 feature instances over the ACE-2005 training set.

For scoring an assignment, each token $x_i$ with argument candidate $e_k$ is assigned a feature vector $f(x, y, i, k)$ (which also includes feature vectors just for trigger assignments in the case of $k=0$). This vector has one slot per feature instance $\phi \in \Phi$, marked $f_\phi(x, y, i, k)$, holding a boolean feature value. A feature value $f_\phi(x, y, i, k)$ is defined to be turned on when $\phi$ is generated by applying
feature type $\phi_f$ to the current $x_i$ and $e_k$ with assignment $y$:

$$
\forall \phi \in \Phi : f_\phi(x, y, i, k) =
\begin{cases} 
1 & \text{if } \phi = \phi_f(x, y, i, k) \\
0 & \text{otherwise}
\end{cases}
$$

This effectively means that $f$ is an indicator vector, choosing only the currently relevant feature instances. When calculating $\text{score}(x, y)$, this indicator vector determines that the slots of weight vector $w$ that will be summed will be the slots of feature instances that are relevant to the current $x_i$, $e_k$ and $y$.

We can now conclude with the full calculation of $\text{score}(x, y)$:

$$
\text{score}(x, y) = w \cdot f(x, y) =
\sum_{i=1}^{s} \sum_{k=0}^{m} w \cdot f(x, y, i, k) =
\sum_{i=1}^{s} \sum_{k=0}^{m} \sum_{\phi \in \Phi} w_\phi \cdot f_\phi(x, y, i, k) =
\sum_{i=1}^{s} \sum_{k=0}^{m} \sum_{\phi \in \Phi : \phi = \phi_f(x, y, i, k)} w_\phi
$$

### 2.3.3 Feature Types

Feature types are either local or global, and either trigger- or argument-focused. The full list of feature types is given in Table 2.1.

A **local feature** records some phenomenon of the text of a trigger/argument candidate, possibly also using its context in text. For instance, a trigger local feature can record the lemmas or the parts-of-speech of a trigger candidate and its preceding token. An argument local feature can record the dependency path between the argument candidate and its trigger in a dependency graph, or record the length of such path.

A **global feature**, on the other hand, can also address other labels in the current assignment, thus connecting between different triggers and arguments in the sentence. For instance, a trigger global feature can record the bigram of trigger labels (event types), possibly alongside the dependency path between them. An
<table>
<thead>
<tr>
<th>Category</th>
<th>Type</th>
<th>Feature Description</th>
</tr>
</thead>
</table>
| Trigger  | Local  | 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   | Info   | 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 | Local  | 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 |
| Trigger  | Global | -                                                                                                                                                                                                                   |
|          |        | 1. bigram of trigger types occurring 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 combined with their types |
| Argument | Global | -                                                                                                                                                                                                                   |
|          |        | 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 participates as an argument in two event mentions in the same sentence  
6. features about two argument 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 combined with the event subtype |

Table 2.1: Local and global feature types
argument global feature can record the dependency path between two argument candidates that share the same role for the same trigger, or record some information over an argument candidate that has a role for two different triggers in a sentence.

2.3.4 Training

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_t$: Training set ${(x, \hat{y})}$</td>
<td>model $w$: weight vector $(w_\phi \mid \phi \in \Phi)$</td>
</tr>
<tr>
<td>$D_d$: Development set ${(x, \hat{y})}$</td>
<td></td>
</tr>
<tr>
<td>$T$: number of iterations</td>
<td></td>
</tr>
</tbody>
</table>

1. $w \leftarrow 0, W \leftarrow \emptyset$
2. for $\tau \leftarrow 1 \ldots T$ do
3. for each $(x, \hat{y}) \in D_t$ do
4. $y \leftarrow \text{beamSearch}(x, \hat{y}, w)$
5. if $y \neq \hat{y}$ then
6. $w \leftarrow w + f(x, \hat{y}) - f(x, y)$ /*early-update*/
7. $W \leftarrow W \cup \{w\}$
8. $w' = \arg\max_{w \in W} \text{score}(D_d, w)$
9. return $w'$

Algorithm 2: General perceptron training with beam search, following [Huang et al.] (2012). $\text{score}(D_d, w)$ runs beam search on each instance in the development set, calculates performance of both trigger and argument extraction, and returns the harmonic mean of both $F_1$ values.

The system utilizes a structured perceptron, an extension to the standard linear perceptron designed for structured prediction, as was proposed in (Collins, 2002). The perceptron learns the feature weights model $w$ in an online fashion, as illustrated in Algorithm 2.

Let $D_t = \{(x, \hat{y})\}$ be the set of training instances. In each iteration $\tau$, the algorithm finds the highest scoring assignment $y$ for $x$, by calling the decoding method $\text{beamSearch}$ (Algorithm 1). If $y$ is incorrect (i.e. not identical to $\hat{y}$), the
weights of $w$ are updated as follows:

$$w = w + f(x, \hat{y}) - f(x, y)$$

As mentioned, we utilize the early-update strategy. Collins and Roark (2004) proposed the early-update idea, and Huang et al. (2012) later proved its convergence and formalized a general framework which includes it as a special case. In each step of the beam search, if the gold standard assignment $\hat{y}$ falls out from the beam, then the top result in the beam is returned for early update. To reduce overfitting, an “averaged perceptron” (Collins, 2002) is used - weights in $w$ are averaged along iterations $\tau$.

In order to ensure the quality of the chosen model, we split a small set of instances from the training set $D_t$ into a development set $D_d$. This set is used to evaluate each model $w$ learned in each iteration $\tau$, and eventually choose the best one. This is done by running beamSearch on all development instances, and then evaluating the performance relative to their gold labels. This is done both for trigger extraction and for argument extraction, since we wish to choose the best $w$ for both subtasks. Finally, we calculate the harmonic mean of both the trigger extraction $F_1$ and the argument extraction $F_1$, and choose the $w$ that yielded the highest mean out of all iterations.

As this training scheme learns weights for feature instances $\phi \in \Phi$ that are dependent on event types, it requires training data for each event type that the system is expected to extract. Producing large amounts of annotated data requires extensive effort, and this motivates a line of research that attempts at reducing the amount of annotated data required for training. In our work, we propose a setting that does not require any annotated documents for new target event types.
Chapter 3

Seed-Based Trigger Labeling

This chapter details the main contribution of our work, addressing only trigger labeling. We first describe our proposed problem setup, which suggests that information of target event types would come from manually-constructed seed lists, instead of from annotated training documents. We then describe our specific method, which relies on the structured perceptron of Li et al. (2013), but with altered labels, scoring scheme and feature types. Finally, we specify the experiments we conducted with our system, and show our improved performance over the baseline fully-supervised system of Li et al.

3.1 Problem Setup

As mentioned in section 2.3, the fully-supervised system’s training phase learns characteristic information of target events by recording it in classification feature instances $\phi \in \Phi$. The information is taken from annotated training documents, which contain labeled event mentions of all target event types. These training documents are manually built by annotators, which is a very costly process. This motivates us in defining our goal, as follows.

**Research goal:** Allow a system to operate on new target event types without requiring annotated data for them, utilizing only information available in event descriptions.
Table 3.1: Seed lists of several event types collected from Section 5 of the ACE-2005 guidelines

We will walk through the steps required for achieving this goal, thus defining our problem setup.

3.1.1 Seed Lists

To achieve our goal, we must provide the system with some characteristic information about target event types that is not acquired from annotated training documents. For that we turn to the annotation guidelines, previously discussed in section 2.1.1. As we recall, the event descriptions in the guidelines embed various event trigger examples, both in the verbal specifications and in the illustrating sentences. This is clearly visible in the top part of Figure 2.1 (page 7).

In our proposed setup, all triggers in a given event description are collected into a seed list of that event type. This seed list is later provided as input to our trigger labeling method (detailed in section 3.2). As an example, triggers from Figure 2.1 are included in the Convict seed list, shown in Table 3.1. Seeds shall be included in their lemmatized form (which may occasionally allow multi-word expressions such as “blow up”). This will allow the system to use knowledge resources such as WordNet (Fellbaum 1998) for expansion, thus reaching great lexical variety. This also means that seed lists need not be exhaustive, but rather only give representative examples for the event’s scope.
The information in a seed list is equivalent to that in the trigger portion of the annotation guidelines. Thus, when a new target event type is introduced to the system, the same effort that would have been used for building the trigger annotation guidelines is used for building the seed list. This way, we eliminate the need for manual annotation of the event, saving much effort.

3.1.2 Features

Once the system is provided with a seed list, it must utilize it for trigger labeling. As we are assuming that our system is based on a classifier, an appealing way of utilizing the information for classification is via classification features. We propose that features will possess these two fundamental properties:

1. *Measuring Trigger Similarity:* Classification features would compute different similarity measures between a candidate trigger in a classified text and the seed list. Candidate triggers that will be recognized as similar to the seeds would be labeled as triggers by the system. Different feature types could implement different mechanisms of measuring such similarity. For example, one feature type can check if the candidate trigger is a WordNet synonym of any of the seeds in the seed list. If synonymity is found, it is a strong indication that the candidate trigger is indeed a trigger of the respective event type.

2. *Being Event-Independent:* A trained model is designed to work on new target event types, that are not known during the system training phase. Therefore, the feature instances $\phi \in \Phi$ must be completely event-independent. This is in contrast to the fully-supervised system, which creates distinct feature instances per each trigger label $y_i \in \mathcal{L} \cup \{\bot\}$. Event-independence also has the potential of reducing the size of the feature space, as multiplying features for different label types is a major contributor to the large feature space in the fully-supervised system.
3.1.3 Training

Our event-independent similarity features require weights, which must be learned during the training phase. Training requires annotated documents as gold standard, yet we previously required that new target event types will not need to have annotated data. Therefore, for the training phase we require a small number of *training event types* - a small set of arbitrary event types that represent future event types, based on which we shall learn feature weights.

Additionally, since the training algorithm (as described in section 2.3.4) uses development documents to evaluate the learned models, we shall also require a small set of *development event types*, which will be different events from the training event types. This is in order to maintain complete independence between learning weight vectors and assessing them for choosing the best one, as the algorithm requires.

For each training and development event type we require a small amount of annotated data (training data and development data). For example, in our evaluation we use 3 training event types and 2 development event types, with a total of 30 annotated trigger mentions. This is compared to roughly 5000 annotated mentions used by the baseline fully-supervised system. In this setting, the training phase is required only once during system setup, while no further training is required for each new target event type.

3.1.4 Summary

Figure 3.1 provides a general overview of our approach. In summary, our setup requires:

1. A seed list per new target event type
2. A small number of annotated triggers for few training and development event types, along with their seed lists (at system setup)
3.2 Method

We now describe the method we designed to implement the seed-based approach. To assess our approach, we compare it (in section 3.3) with the common fully-supervised approach, which requires annotated data for each target event type. Therefore, we implemented our system by adapting the trigger classification part of the state-of-the-art fully-supervised event extraction system of Li et al. (2013), to which we compare our system’s performance. We modified mechanisms relevant for features and for trigger labels, as described below. This way, the systems are comparable with respect to using the same pre-processing and machine learning infrastructure.

All relevant aspects of that system were described in section 2.3 here we describe the parts we modified, in order to fit the system to our setup and approach.
3.2.1 Labels

Whereas the fully-supervised system only operates on event types that were processed during training, our system allows processing for any new target event type. Since future event types are not known during the training phase, we cannot train a multi-class classifier like in the fully-supervised system. Instead, we will have a binary classifier, supporting trigger labels \( \{\top, \bot\} \) (read top and bottom). Every trigger candidate \( x_i \) is given by the system a label \( l \) that either means that \( x_i \) is a trigger \( (l = \top) \) or not \( (l = \bot) \), for an event type that is currently being considered.

To support this setting, each input sentence must be processed only for a single target event type at a time. Let \( \mathcal{T} \) be a set of target event types, for which we want to label our input documents. We classify each input sentence multiple times, once for each target event type \( t \in \mathcal{T} \). Thus when a trigger candidate \( x_i \) is labeled \( l = \top \), this means that the classifier determines that \( x_i \) is a trigger of the current target event type \( t \). For instance \( x_i = \text{“visited”} \) labeled as \( \top \) when classifying for \( t = \text{Meet} \), means that \( x_i \) is labeled as a \text{Meet} trigger. This is performed both during the training phase (and then \( \mathcal{T} \) contains the training event types), and during the labeling phase (and then \( \mathcal{T} \) contains the new target event types).

Since we label each token independently for each event type \( t \), multiple labels may be assigned to the same token. If a single-label setting is required, standard techniques can be applied, such as choosing a single random label, or the highest scoring one.

3.2.2 Assignment Scoring

The decoding process in our system behaves similarly to the one in the fully-supervised system, as described in section 2.3.1. However, one main difference from that process is the way scores are calculated for assignments.

We recall that in the fully-supervised system, when a feature type \( f \) is applied to token \( x_i \) it generates some textual or numerical value \( f(x, i) \) that becomes part of feature instance \( \phi \). For example, applying \( f_1 = \text{TriggerBackBigram} \) to \( x_i = \text{“He perished at noon.”} \) generates \( f_1(x, i) = (\text{“He”}, \text{“perished”}) \), and the generated feature instance is \( \phi_1 = \langle \text{TriggerBackBigram}, (\text{“He”}, \text{“perished”}), \text{Die} \rangle \) (equation 2.3.2.1). The full calculation of \( \text{score}(x, y) \) is detailed in equation 2.3.2.3.
In order to implement our approach, we make the following modifications to the fully-supervised system:

1. Since \( x \) is processed separately for each event type \( t \in \mathcal{T} \), we apply feature type \( f \) not just to \( x_i \), but also to the seed list of \( t \), marked \( \Psi_t \). Thus we get \( f(x_i, \Psi_t) \).

2. Each feature type \( f \in \mathcal{F} \) represents a *similarity examining mechanism*. This means that each feature type implements some method to examine similarity between a given token \( x_i \) and the seeds in seed list \( \Psi_t \). For example, feature type \( f = \text{Synonym} \), checks whether two terms are synonyms according to WordNet. Specifically, when applied to a trigger candidate token \( x_i \) and seed list \( \Psi_t \), \( f(x_i, \Psi_t) \) checks whether \( x_i \) is a synonym of any of the seeds in the seeds list. This means that we expect \( f(x_i, \Psi_t) \) to return a numerical value that provides some indication of the similarity between \( x_i \) and \( \Psi_t \) according to \( f \) (unlike in the fully-supervised system, where \( f \) yielded some textual value, like (“He”, “perished”)). For simplicity, we use only similarity examining mechanisms that output a boolean value: 1 if similarity was found, or 0 if not. So in our example, \( f = \text{Synonym} \) would return the value 1 if \( x_i \) is a synonym of at least one seed. Future research may also experiment with a richer representation of similarity, for instance allowing a continuous value for \( f(x_i, \Psi_t) \), representing different degrees of similarity.

3. Feature instance \( \phi \) is comprised solely from a pair \( \langle f, l \rangle \), combining a similarity examining measure \( f \in \mathcal{F} \) and label \( l \in \{ \top, \bot \} \). For example, feature instance \( \phi_6 = \langle \text{Synonym}, \bot \rangle \) combines feature type \( f = \text{Synonym} \) and label \( l = \bot \) (which means that the current token is not a trigger). Compare this with a corresponding example of \( \phi \) in the fully-supervised system (equation 2.3.2.1). We use \( \phi_f \) to mark a feature instance’s feature type, and \( \phi_l \) to indicate a feature instance’s label.

4. Feature value \( f_{\phi}(x, i, y, t) \) is defined as \( \phi_f(x_i, \Psi_t) \), but only if \( y_i = \phi_l \) (the assignment of token \( x_i \) is equals the label of feature instance \( \phi \)). This, and other concepts detailed in this section, are exemplified in Table 3.2.
We shall put these new definitions formally:

\[ \Phi = \{(f, l) \mid f \in F, l \in \{\top, \bot\}\} \]

\[ \forall \phi \in \Phi : f_\phi(x, y, i, t) = (\phi_f(x_i, \Psi_t) = \text{True}) \land (\phi_l = y_i) \]

The first formula defines that all feature instances \( \phi \in \Phi \) will simply be comprised of all pairs of feature type and label \((f, l)\). The second formula means that a feature instance’s value in the feature vector would be 1 iff the feature instance’s label equals the label assigned to the current trigger candidate token \((\phi_l = y_i)\) and similarity examining mechanism \(f\) indeed found similarity between \(x_i\) and the seed list \((\phi_f(x_i, \Psi_t) = \text{True})\).

Note that since there are only two trigger labels, and a handful of feature types, we get a very small space of feature instances. In our system this amounts to 8 feature instances, as described in Table 3.2.
Following is our system’s calculation of \( \text{score}(x, y, t) \), with an explanation for each line.

\[
\begin{align*}
\text{score}(x, y, t) &= | \text{score of sentence } x \text{ and assignment } y \text{ for event type } t \\
\mathbf{w} \cdot \mathbf{f}(x, y, t) &= | \text{dot product of feature weights } \mathbf{w} \text{ and feature vector } \mathbf{f} \\
\sum_{i=1}^{s} \mathbf{w} \cdot \mathbf{f}(x, y, i, t) &= | \text{sum feature vectors } \mathbf{f} \text{ over each token } i \\
\sum_{i=1}^{s} \sum_{\phi \in \Phi} \mathbf{w}_\phi \cdot \mathbf{f}_\phi(x, y, i, t) &= | \text{sum feature values } \mathbf{f}_\phi \text{ over each feature instance } \phi \\
\sum_{i=1}^{s} \sum_{\phi \in \Phi: \phi_l = y_i \land \phi_f(x_i, \Psi_t) = \text{True}} \mathbf{w}_\phi &= | \text{sum only weights } \mathbf{w}_\phi \text{ with feature value of 1, which are } \\
&\quad \text{the ones of feature instances } \phi \text{ that have label } y_i, \\
&\quad \text{and found similarity between token } x_i \text{ and seed list } \Psi_t
\end{align*}
\]

Similarly to the fully-supervised system, we can see that feature vector \( \mathbf{f} \) is an indicator vector, determining which slots in \( \mathbf{w} \) will be summed. It will be only slots \( \mathbf{w}_\phi \) of feature instances \( \phi \) where the label \( l \) in \( \phi \) is the one assigned to the current candidate trigger \( y_i \), and only if \( \phi_f \) finds similarity between the current candidate trigger and the current seed list.

### 3.2.3 Feature Types

We implement a basic set of feature types that examine different aspects of similarity between a candidate token \( x_i \) and a seed list \( \Psi \) (Table 3.3). Most features rely on the lexical database WordNet (Fellbaum, 1998). We begin with addressing the general mechanism of the features, and then describe each feature type in detail.

For simplicity, our features are disjunctive, or “any”-based: if similarity is found with at least one seed \( \psi \in \Psi \), the feature determines that similarity is found with the entire list. More sophisticated schemes could of course be applied, such as requiring similarity to be found with some larger fixed amount of seeds, or
Table 3.3: Similarity feature types implemented in the system, using only the WordNet knowledge resource.

<table>
<thead>
<tr>
<th>Feature (f)</th>
<th>Event (t)</th>
<th>Seed (ψ)</th>
<th>Candidate Token (x_i)</th>
<th>Relation (r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Same Lemma</td>
<td>Die</td>
<td>kill</td>
<td>... on killing them ...</td>
<td>-</td>
</tr>
<tr>
<td>2. Synonym</td>
<td>Start-Org</td>
<td>launch</td>
<td>... was founded later ...</td>
<td>Synonym</td>
</tr>
<tr>
<td>3. Hypernym</td>
<td>Attack</td>
<td>conflict</td>
<td>... mass uprising in ...</td>
<td>Hypernym</td>
</tr>
<tr>
<td>4. Similarity Relations</td>
<td>Transport</td>
<td>travel</td>
<td>... to arrive after ...</td>
<td>Entailment</td>
</tr>
</tbody>
</table>

When choosing synsets to use, we take all existing synsets for each candidate token $S_{x_i}$, but only the first synset for each seed $S_{ψ}$. The first synset of a WordNet entry is the most frequent sense of the term in WordNet’s reference corpus, and thus conceptually represents a more typical use of the term. This configuration was found in experiments to be the most effective in our setting.

The full list of feature types in our system is brought in Table 3.3, providing a concrete example for each feature type. We next describe each feature type in more detail:
1. **Same Lemma**: This feature checks if the candidate token and the seed have the same lemma. This is the only non-WordNet feature in the system, and thus has a slightly modified definition for \( f(x_i, \Psi_t) \): it returns 1 iff \( \exists \psi \in \Psi_t \) such that \( \text{lemma}(x_i) = \text{lemma}(\psi) \). This feature is intended to be a high-precision-low-recall feature, as the same lemma would usually represent the same concept (and thus be an event trigger), yet no semantic variability would be addressed.

2. **Synonym**: This feature captures synonymity between the candidate token and the seed. This aims at addressing slightly more semantic variability, on the expense of precision.

3. **Hypernym**: This feature checks for hypernymy, meaning that the seed refers to some type that is more general than the candidate token. To check that, we are using two relevant WordNet relations - *hypernym*, which checks if the candidate token is a subtype of the seed type, and *instance hypernym*, which checks if the candidate token is an instance of the seed type. As opposed to the first two features, this feature is asymmetric.

4. **Similarity Relations** This feature utilizes a set of WordNet relations, each representing some type of similarity:
   - Synonym
   - Hypernym
   - Instance Hypernym
   - Part Holonym
   - Member Holonym
   - Substance Meronym
   - Entailment

   The combination of these relations can cover a wide range of semantic variability. As the other extreme of the first feature, the current feature aims at low-precision-high-recall.

   The features **Hypernym** and **Similarity Relations** attempt at capturing a wide range of seed-candidate pairs, using these two mechanisms: (1) Allowing 2 levels
Table 3.4: Split of the ACE-2005 documents, used in our evaluation as well as previous evaluations: \cite{Ji:2008, Liao:2010, Li:2013}.

<table>
<thead>
<tr>
<th>Category</th>
<th>Number of Documents</th>
<th>Number of Sentences</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Training</strong></td>
<td>529</td>
<td>14,840</td>
</tr>
<tr>
<td><strong>Development</strong></td>
<td>30</td>
<td>863</td>
</tr>
<tr>
<td><strong>Test</strong></td>
<td>40</td>
<td>672</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>599</td>
<td>16,375</td>
</tr>
</tbody>
</table>

of transitivity. For example, a hypernym of a hypernym. Theoretically, higher transitivity levels would still preserve similarity, yet in practice this situation tends to be too noisy and introduce many false positives. (2) In addition to checking the candidate token, we also check all of its WordNet derivations. This allows us, for instance, to switch between a noun and a verb, and vice versa.

Note that WordNet is the only knowledge resource we use. This could be potentially extended to further resources, such as the Nomlex nominalizations lexicon \cite{Macleod:1998}, Brown clusters \cite{Brown:1992, Miller:2004, Sun:2011}, or paraphrase databases (like in \cite{Ganitkevitch:2013}).

3.3 Evaluation

3.3.1 Setting

We evaluate our seed-based approach in comparison to the fully-supervised approach implemented by \cite{Li:2013} (described in section 2.3), addressing only the trigger labeling portion. We thus use their system and published results as a baseline for our experiments.

To maintain comparability, we evaluate our system over the same 33 event types defined by the ACE-2005 event extraction task, and the same 599 documents provided with the task. We also use the same split of these documents (detailed in Table 3.4) as previously used by \cite{Ji:2008, Liao:2010}.
As for additional parameters of the system, we use the same beam size $K = 4$, and the same number of iterations $T = 20$. Li et al. (2013) showed that these parameters are sufficient for yielding high results. Furthermore, as Li et al. (2013) address different evaluation settings with regard to whether input argument candidates are gold-standard or predicted by a third-party tool, we can ignore this entire aspect, as we label only triggers and no arguments.

However, some evaluation settings differ: Li et al. (2013) train a multi-class model for all 33 ACE-2005 event types, and classify all tokens in the test documents into these event types. Our approach, on the other hand, trains an event-independent binary classifier, while testing on new event types (one at a time) that are different from those utilized for training. We next describe how this setup is addressed in our evaluation. Table 3.5 displays a portion of our evaluation runs, thus exemplifying the following principles:

**Per-Event Classification** To label the test documents for all 33 event types, we classify each token in the test documents once for each test event type. Note that Li et al. (2013) produced a single-label classification, and as mentioned in section 3.2.1, our classification may be multi-label. To maintain comparability, we randomly choose a single label in every case that more than one label was assigned. This is a rare case, happening only for 7 tokens in our entire input, with only two labels assigned per token (and not more) in these 7 cases.

**Event Types for Training** When we label for a test event type $t$, we use a model that was trained on different pre-selected training and development event types. Since we need to label for all event types, we cannot use the same model for testing them all, since then the event types used to train this model could not be tested. Thus, when labeling for each test event type $t$, we use a model trained on 5 randomly chosen event types, which different than $t$: 3 training event types, and 2 development event types.

Additionally, to avoid a bias caused by a particular random choice of event types, we build 10 different models for each $t$, each time choosing a different set of 5 event types for training and development. Then, we perform labeling of the
Table 3.5: 20 out of 330 evaluation runs. Each run has a global identifier (G) and a per-test-type identifier (T). For each event type or list of event types we specify the amount of annotated triggers we use (tr), within the corresponding documents. For the test event type we use all available annotated mentions in the test documents, whereas for the training and development event types we use fixed sampled amounts - 24 and 6 respectively. Finally, we show the $F_1$ score of each run, as a “sneak-peek” to our results. Note the low variability of $F_1$ for each test event type across different runs.
test documents for $t$ 10 times, once by each model. When measuring performance we compute the average of these 10 runs for each $t$, as well as the variance within these runs.

This brings us to a total of 330 runs - 10 runs for each of the 33 event types. Table 3.5 gives 20 of these runs as an example.

**Annotated Triggers** Training and development event types require annotated triggers from the training documents. To maintain consistency between different sets of event types for training, (which have different amounts of triggers in the training documents) we use a fixed total of 30 annotated triggers for each set of event types.

The 30 annotated triggers are split - 24 triggers of the 3 training event types (from the training documents), and 6 triggers of the 2 development event types (from the development documents). All of these annotated triggers, taken from multiple event types, are randomly chosen in a way that preserves their original distribution in the dataset. For example, if event types $A$ and $B$ are used as the 2 development event types, and $A$ has twice as many mentions in the development documents as $B$, then the 6 development triggers that will be randomly chosen will have 4 triggers of $A$, and 2 of $B$.

The amounts of 5 event types for training and 30 annotated triggers were chosen to demonstrate that even such small amounts, requiring little manual effort at system setup, can yield high performance. In our experiments, larger training didn’t improve results, possibly due to the small number of features.

**Seed Lists** To build the seed lists for all event types, we manually extracted all triggers mentioned in Section 5 of the ACE-2005 guidelines, as described in section 3.1.1. Our full seed lists are detailed in Table 3.6. The lists have 4.2 seeds per event type on average, which is fairly small. For comparison, each event type has on average 46 distinct trigger terms in the training corpus used by the fully-supervised method.
<table>
<thead>
<tr>
<th>#</th>
<th>Event Type</th>
<th>Seed List</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Acquit</td>
<td>acquit</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Appeal</td>
<td>appeal</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Arrest-Jail</td>
<td>incarcerate, arresting, arrest, jail, imprison, custody</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Attack</td>
<td>coup, firing, shot, battle, bomb, bombing, blow, attack, clash, tabbing, terrorism, terrorist activity, war, conflict, explosion, stab, throw, shoot, violence, fighting, explode, go off, gunfire</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Be-Born</td>
<td>born, birth</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Charge-Indict</td>
<td>charge, indict, indictment, accuse, charged</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>Convict</td>
<td>convict, guilty</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>Declare-Bankruptcy</td>
<td>bankrupt, bankruptcy</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>Demonstrate</td>
<td>strike, demonstrate, walkout, rioting</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>Die</td>
<td>death, dying, slay, die, assassination, dead, killing, suicide, kill, murder, assassinate, fatal, deceased</td>
<td>13</td>
</tr>
<tr>
<td>11</td>
<td>Divorce</td>
<td>divorce, divorced</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>Elect</td>
<td>elected, elect, election</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>End-Org</td>
<td>fold, shut down</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>End-Position</td>
<td>layoff, retired, fire, retire, sack, succeed, lay off</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>Execute</td>
<td>execute</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>Extradite</td>
<td>extradition, extradite</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>Fine</td>
<td>fine</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>Injure</td>
<td>injured, harm, injure, hospitalize, hurt, disabled, wounded</td>
<td>7</td>
</tr>
<tr>
<td>19</td>
<td>Marry</td>
<td>marry, married, marriage</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>Meet</td>
<td>conference, visit, summit, talks, meet, meeting</td>
<td>6</td>
</tr>
<tr>
<td>21</td>
<td>Merge-Org</td>
<td>merger, merge</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>Nominate</td>
<td>nominate, nomination</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>Pardon</td>
<td>pardon</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>Phone-Write</td>
<td>write, phone, communicate</td>
<td>3</td>
</tr>
<tr>
<td>25</td>
<td>Release-Parole</td>
<td>release, released, parole, free</td>
<td>4</td>
</tr>
<tr>
<td>26</td>
<td>Sentence</td>
<td>sentence</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>Start-Org</td>
<td>launch</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>Start-Position</td>
<td>start office, appoint, hire, take over, succeed, become</td>
<td>6</td>
</tr>
<tr>
<td>29</td>
<td>Sue</td>
<td>sue, lawsuit, suit</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>Transfer-Money</td>
<td>give, pay, lend, borrow, donation, aid, donate</td>
<td>7</td>
</tr>
<tr>
<td>31</td>
<td>Transfer-Ownership</td>
<td>acquired, purchase, buy, acquire, takeover</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>Transport</td>
<td>return, travel, move, leave, moving, take, flee, transport</td>
<td>8</td>
</tr>
<tr>
<td>33</td>
<td>Trial-Hearing</td>
<td>try, trial, court proceeding, hearing</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3.6: Seed lists manually collected for our evaluation from Section 5 of the ACE-2005 guidelines
3.3.2 Results

Table 3.7 shows our system’s performance over the total 330 runs. We show micro-average precision, recall and $F_1$. We report micro-average as it is the typical measure for this task. Macro-average results are a few points lower for our system and for the system of Li et al. (2013), maintaining similar relative difference.

<table>
<thead>
<tr>
<th></th>
<th>Micro-Avg. (%)</th>
<th>$F_1$ Variance (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec</td>
<td>Rec</td>
</tr>
<tr>
<td>Seed-Based</td>
<td>80.6</td>
<td>67.1</td>
</tr>
<tr>
<td>Li et al. (2013)</td>
<td>73.7</td>
<td>62.3</td>
</tr>
<tr>
<td>Ji and Grishman(2008)</td>
<td>67.6</td>
<td>53.5</td>
</tr>
</tbody>
</table>

Table 3.7: Seed-Based performance compared to fully-supervised systems, plus average and variance of $F_1$ variance (%) over the 10 test runs per test event type.

Additionally, Table 3.7 addresses the variance of $F_1$ within the group of 10 runs of each test event type. We specify both the average variance over all run groups, and also the variance’s variance across all run groups. The very low variance indicates that the system’s performance does not depend much on the choice of training and development event types.

We compare our system’s performance to the published trigger classification results of the baseline system of Li et al. (2013) (its globally optimized run, when labeling both triggers and arguments). We also compare to the sentence-level system in Ji and Grishman (2008) which uses the same dataset split. Our system outperforms the fully-supervised baseline by 5.7% $F_1$, which is statistically significant (using two-tailed Wilcoxon test, with $p < 0.05$). This shows that there is no performance hit for the seed-based method on this dataset, even though it does not require any annotated data for new tested events, thus saving costly annotation efforts.
3.3.3 Error Analysis

We analyze our results to discover the reasons for different kinds of errors made by our system. We measure the effect each of the different reasons had over the general results, thus determining our system’s most prominent hindering factors, as well as potential future directions to overcome them.

Typically to a binary classifier scenario such as ours, errors are divided to false-positives and false-negatives. For each of these error types, we randomly sampled 100 wrongly labeled instances from the system output, and manually analyzed the reason for each error.

<table>
<thead>
<tr>
<th></th>
<th>Reason</th>
<th>Mistake of</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Wrong candidate token synset</td>
<td>System</td>
<td>39%</td>
</tr>
<tr>
<td>2</td>
<td>Wrong seed synset</td>
<td>System</td>
<td>21%</td>
</tr>
<tr>
<td>3</td>
<td>Different word choice for trigger</td>
<td>System</td>
<td>17%</td>
</tr>
<tr>
<td>4</td>
<td>Candidate token too general</td>
<td>System</td>
<td>14%</td>
</tr>
<tr>
<td>5</td>
<td>Missing gold annotation</td>
<td>Gold</td>
<td>9%</td>
</tr>
</tbody>
</table>

Table 3.8: Distribution of reasons for false-positive errors. Most error reasons are caused by the system, yet one originates in the gold standard labeling.

Table 3.8 details the distribution of reasons for false-positive errors. We shall describe each reason in detail:

1. **Wrong candidate token synset**: As mentioned in section 3.2.3, a candidate token $x_i$ is considered a trigger if it has a synset $S_{x_i}$ that is related to a seed synset $S_{\psi}$ via relation $r$. We mark these specific synsets as $S^r_{x_i}$ and $S^r_{\psi}$, as they are the ones that have $r$ relating them. On the other hand, we know that $x_i$ has a specific meaning in the context of the text sentence $x$. This meaning can be mapped to a synset as well, which we will mark $S^x_{x_i}$, as it is chosen by sentence $x$.

In the current error reason, $S^r_{x_i} \neq S^x_{x_i}$. This means that the candidate token is related to the seed list via a meaning that is different from its meaning in
the context of the sentence. This results in an erroneous linking of the token to the seed list, and thus a wrong labeling. Some examples for that are given in Table 3.9.

<table>
<thead>
<tr>
<th>Event</th>
<th>Sentence</th>
<th>Sentence Synset $S^x_{x_i}$</th>
<th>Seed $\psi$</th>
<th>Relation Synset $S^r_{x_i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack</td>
<td>... the war, while <strong>opposed</strong> by most Russians, was...</td>
<td>“be against; express opposition to”</td>
<td>fighting</td>
<td>“fight against or resist strongly”</td>
</tr>
<tr>
<td>Attack</td>
<td>If the resolution is not <strong>passed</strong>, Washington will...</td>
<td>“make laws, bills, etc. or bring into effect by legislation”</td>
<td>throw</td>
<td>“throw (a ball) to another player”</td>
</tr>
</tbody>
</table>

Table 3.9: Examples for false-positive Wrong candidate token synset

A prominent direction in addressing this type of error is via word-sense disambiguation (WSD). WSD aims at choosing the intended sense of a word in a given context. State-of-the-art WSD systems, such as (Agirre et al., 2014) and (Ponzetto andNavigli, 2010), achieve high performance that can be useful for our goal.

2. **Wrong seed synset**: Similarly to $x$, having two relevant kinds of synsets $S^r_{x_i}$ and $S^x_{x_i}$, we discuss two synset kinds of seed $\psi$: $S^r_{\psi}$ is the aforementioned sense that $S^r_{x_i}$ relates to via relation $r$. Alongside it, $S^t_{\psi}$ is the sense that links $\psi$ to type $t$ - when seed list $\Psi_t$ was manually built, each seed was added since at least some of its senses are relevant as triggers of $t$.

In the current error reason, $S^r_{\psi} \neq S^t_{\psi}$. This means that the seed is related to the candidate token via a meaning that is different from the meaning that links it to the seed list of $t$. This results in an erroneous linking of the token to the seed list, and thus a wrong labeling. Some examples for that are given in Table 3.10.
Meet ... are widely seen here as a blow to Moscow ...

EndPos ... for Erdogan, who won a parliamentary seat in by-elections ...

<table>
<thead>
<tr>
<th>Event</th>
<th>Sentence $x$</th>
<th>Seed $\psi$</th>
<th>Relation Synset $S^\psi_{\psi}$</th>
<th>Type Synset $S^\psi_{\psi}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meet</td>
<td>... are widely seen here as a blow to Moscow ...</td>
<td>visit</td>
<td>“go to see a place, as for entertainment”</td>
<td>“come to see in an official or professional capacity”</td>
</tr>
<tr>
<td>EndPos</td>
<td>... for Erdogan, who won a parliamentary seat in by-elections ...</td>
<td>succeed</td>
<td>“attain success or reach a desired goal”</td>
<td>“be the successor (of)”</td>
</tr>
</tbody>
</table>

Table 3.10: Examples for false-positive Wrong seed synset

Similarly to the previous type of error, a state-of-the-art WSD system could address this error type as well. Handling these two types of errors means addressing a total of 60% of the system’s false-positives, which leads us to the conclusion that applying WSD has great potential for improving our system’s performance.

3. **Different word choice for trigger**: Sometimes multiple tokens in a sentence can be the trigger of an event, leaving it for the annotator to decide which token to annotate. This is usually done by judging which of the tokens more prominently represents the event. In the current error reason, the system labeled one of those potential tokens (which in other contexts could have been a valid trigger), which was not chosen as the trigger by the annotator. Some examples for that are given in Table 3.11.

<table>
<thead>
<tr>
<th>Event</th>
<th>Sentence $x$ (with system’s trigger and gold trigger)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meet</td>
<td>... the US diplomat said after the informal summit talks.</td>
</tr>
<tr>
<td>Die</td>
<td>The US and Britain suffered their first casualties Thursday and Friday as they pushed into Iraq, with two U.S. Marines killed in combat ...</td>
</tr>
</tbody>
</table>

Table 3.11: Examples for false-positive Different word choice for trigger

The responsibility for this type of error can also be partially addressed to the annotation methodology of the gold standard, since it can be viewed as somewhat incomplete. Had the false-positives of this type been somehow
labeled in the gold data, it is possible that some of them could have been considered as a correct labeling.

4. **Candidate token too general**: Here the system labels a token that is indeed related to event type $t$, but refers to it only generally, and does not indicate an event instance. Some examples for that are given in Table 3.12.

<table>
<thead>
<tr>
<th>Event $t$</th>
<th>Sentence $x$</th>
<th>Why marked trigger is too general</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack</td>
<td>... ordinary Iraqis still felt unsafe on the street with <strong>gunfire</strong> rife and police a near invisible presence.</td>
<td>No specific gunfire incident, rather stating that there are generally many gunfire incidents.</td>
</tr>
<tr>
<td>Meet</td>
<td>... the DPRK should scrap its nuclear program before <strong>dialogue</strong> ...</td>
<td>Refers to any future dialogue, not a specific meeting.</td>
</tr>
</tbody>
</table>

Table 3.12: Examples for false-positive **Candidate token too general**

One direction for addressing this type of error would be to develop a classifier that could distinguish between mentions of concrete events and mentions of general or hypothetical events.

5. **Missing gold annotation**: It seems that the trigger labeled by the system is valid, and should have been labeled by the annotators (but was missed). Some examples for that are given in Table 3.13.

<table>
<thead>
<tr>
<th>Event $t$</th>
<th>Sentence $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>StartOrg</strong></td>
<td>Reformist MDP members, however, vowed to <strong>launch</strong> a preparatory committee next month ...</td>
</tr>
<tr>
<td><strong>Attack</strong></td>
<td>Highway 80, nicknamed the “Highway of Death” during the 1991 Gulf <strong>War</strong>, ...</td>
</tr>
</tbody>
</table>

Table 3.13: Examples for false-positive **Missing gold annotation**
Table 3.14: Distribution of reasons for false-negative errors. Most error reasons are caused by the system, yet one originates in the gold standard labeling.

To complete the analysis, Table 3.14 details the distribution of reasons for false-negative errors. We shall describe each reason in detail:

1. **No relation connecting candidate token and seeds:** In this error reason, no WordNet relation $r$ exists that links between the candidate text token (any sense of it) and the sense of any of the seeds. More formally, this means that: $\neg\exists(r, S_{x_i}, S_\psi) : S_{x_i} \xrightarrow{r} S_\psi$. This could happen when the seed list is not comprehensive enough, and there are variations for expressing that target even that the seed list doesn’t account for (even via WordNet). Some examples for that are given in Table 3.15.

<table>
<thead>
<tr>
<th>Event $t$</th>
<th>Sentence $x$</th>
<th>Seed List $\Psi_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>EndPos</em></td>
<td>... with former Chinese president Jiang Zeming ...</td>
<td>layoff, retired, fire, retire, sack, succeed, lay off</td>
</tr>
<tr>
<td><em>Transport</em></td>
<td>... will get permission to deploy troops in the country ...</td>
<td>return, travel, move, leave, moving, take, flee, transport</td>
</tr>
</tbody>
</table>

Table 3.15: Examples for false-negative No relation connecting candidate token and seeds

In order to address this issue, it is possible to develop a more thorough methodology for constructing the seed lists, one which would aspire to have them include a plethora of variations that express the target event.

2. **Wrong gold annotation:** Here a token was manually labeled by the annotators as a trigger of some event type, even though it was not supposed to
according to the ACE annotation guidelines. This could be in violation of general annotation rules, or ones that are specific to the relevant event type. Some examples for that are given in Table 3.16.

<table>
<thead>
<tr>
<th>Event t</th>
<th>Sentence x</th>
<th>Why the token should not have been annotated as a trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack</td>
<td>... believed to have Scud missiles capable of <strong>reaching</strong> Israel ...</td>
<td>No mention of a specific attack or even a planned one, just weapon capabilities. Violates the general rule of annotating only events that are mentioned.</td>
</tr>
<tr>
<td>Meet</td>
<td>Bush signalled that ..., <strong>inviting</strong> his “good friend” Putin to his weekend retreat ...</td>
<td><strong>Meet</strong> events should refer to the actual talks, not just invitations to them. Violates an event-type-specific rule.</td>
</tr>
</tbody>
</table>

Table 3.16: Examples for false-negative Wrong gold annotation

3. **Coreference**: While usually a trigger would be semantically related to its event type, there are exceptions to this rule, most notably in the case of coreference. In this case, the trigger merely refers to an event mention that is brought in whole somewhere else in the text, by having a token such us “it”, “this” or “that”. Our system is not designed to handle such cases, as it always relies on semantic similarity between the trigger and the event type. Some examples for that are given in Table 3.17.

<table>
<thead>
<tr>
<th>Event t</th>
<th>Sentence x</th>
<th>Referred event mention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport</td>
<td>But <strong>this</strong> is also conditional on police approval.</td>
<td>Prison authorities have given the nod for Anwar to be <strong>taken</strong> home ...</td>
</tr>
<tr>
<td>Marry</td>
<td>... yeah, <strong>it</strong> wasn’t perfect ...</td>
<td>I know my first <strong>marriage</strong> had a lot of ...</td>
</tr>
</tbody>
</table>

Table 3.17: Examples for false-negative Coreference

This type of error can be addressed using an event coreference resolution system, such as (Lee et al., 2012). Such a system can locate pairs of a referring word and a referred event mention, and use that to label the referring word as a trigger as well.
In conclusion, our error analysis shows various reasons for false-positive and false-negative errors, mostly due to mistakes of the system itself. Various directions for addressing these issues were proposed, such as utilizing further external tools (e.g., WSD and an event coreference resolver), and extending the methodology for building seed lists.

3.4 Related Work

Our work contributes to the broader research direction of reducing annotation for information extraction. One such IE paradigm, including Preemptive IE (Shinya and Sekine, 2006), On-demand IE (Sekine, 2006; Sekine and Oda, 2007) and Open IE (Etzioni et al., 2005; Banko et al., 2007, 2008), focuses on unsupervised relation and event discovery. We, on the other hand, follow the same goal as fully-supervised systems in targeting pre-specified event types, but aim at minimal annotation cost.

Bootstrapping methods (such as Yangarber et al., 2000; Agichtein and Gravano, 2000; Riloff, 1996; Greenwood and Stevenson, 2006; Liao and Grishman, 2010; Stevenson and Greenwood, 2005; Huang and Riloff, 2012) have been widely applied in IE. Most work started from a small set of seed patterns, and repeatedly expanded them from unlabeled corpora. Relying on unlabeled data, bootstrapping methods are scalable but tend to produce worse results (Liao and Grishman, 2010) than supervised models due to semantic drift (Curran et al., 2007). Our method can be seen complementary to bootstrapping frameworks, since we exploit manually crafted linguistic resources which are more accurate but may not cover all domains and languages.

Our approach is perhaps closest to Roth et al., 2009. They addressed a different IE task – relation extraction, by recognizing entailment between candidate relation mentions and seed patterns. While they exploited a supervised recognizing textual entailment (RTE) system, we show that using only simple WordNet-based similarity features and minimal training yields relatively high performance in event trigger labeling.
Chapter 4

Specification-Based Event Extraction

This chapter describes our attempts at expanding our seed-based approach and method, to label both triggers and arguments. This unfortunately yielded relatively poor performance, and thus is included here as negative results. Still, we believe that this task may be achievable, and has great contribution to potential the IE research.

We first describe our extended problem setup, introducing event specifications as an expansion for seed lists. Next we describe our method, with labels and feature types relevant for arguments. We finish with describing the evaluation we performed in this setting, and our achieved results.

4.1 Problem Setup

Our extended problem setup keeps the original goal defined in section 3.1 - allow a system to operate on new target event types without requiring large amounts of annotated data of them. We still wish to replace the information recorded in features $\phi \in \Phi$ in fully-supervised systems with data provided during the labeling phase, and still maintain the concept of event-independent similarity features. The main difference here is the scope of information provided on target events.
### Convict

<table>
<thead>
<tr>
<th>Role</th>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defendant</td>
<td>PER ORG GPE</td>
<td>The convicted agent(s)</td>
<td>A Russian court <strong>convicted</strong> [Pope] Wednesday on espionage charges and sentenced him to 20 years in prison.</td>
</tr>
<tr>
<td>Adjudicator</td>
<td>PER ORG GPE</td>
<td>The judge or court</td>
<td>[A Russian court] <strong>convicted</strong> Pope Wednesday on espionage charges...</td>
</tr>
<tr>
<td>Crime</td>
<td>CRIME</td>
<td>The crime for which the Defendant has been convicted</td>
<td>A Russian court <strong>convicted</strong> Pope Wednesday on [espionage] charges...</td>
</tr>
<tr>
<td>Time</td>
<td>TIME</td>
<td>When the conviction takes place</td>
<td>A Russian court <strong>convicted</strong> Pope [Wednesday] on espionage charges...</td>
</tr>
<tr>
<td>Place</td>
<td>GPE LOC FAC</td>
<td>Where the conviction takes place</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.1: Arguments portion of the ACE-2005 event description of *Convict*

As previously done with seed lists for triggers, information of arguments of target events is collected from the annotation guidelines. An example of an event description from the annotation guidelines was given in Figure 2.1 (page 7). The argument portion of this example is brought here again for convenience (Figure 4.1).

In our full event extraction setting, instead of collecting just trigger seeds into a seed list, we build a more comprehensive *event specification*, or “*spec*”. The spec will contain information from the annotation guidelines that is relevant both for triggers and for arguments. The structure of a spec is defined in Figure 4.2.
We elaborate on the different elements:

- **Spec name**: The name of the event type that the spec belongs to, e.g. *Injure*.

- **List: seeds**: A list of seeds, each is an example for a trigger. Identical to the seed-list from section 3.1.1. For example: *meet, conference, summit* for event type *Meet*.

- **List: arguments**: One entry per argument role of the current event type. Each argument is a complex entry, including internal fields.

- **Role name**: Name of the current argument’s role, signifying how it is related to the event type. For example *Attacker, Instrument or Destination* for event type *Attack*.

- **List: argument types**: A list of all possible types that an argument candidate may have in this role. These can be entity types (such as PER, ORG and GPE), value types (such as CRIME and JOB-TITLE) or TIME (signifying a time-expression), as specified in the ACE annotation guidelines (described in section 2.1.1 page 5).

- **List: argument samples**: A list of examples for arguments. An argument sample is equivalent to a trigger seed. An argument sample may consist of
several words, as arguments are often multi-word expressions (as opposed to triggers, which are usually a single word). For example: man, Shirley, rifle, World Health Organization, coffee shop, Chairman of the Joint Chiefs of Staff, Wednesday afternoon.

- **List: labeled usage samples:** A list of sentences, each containing a single occurrence of the event. The trigger and all arguments are taken from the trigger seed list and argument sample lists in the spec. A sample should be a very basic sentence, that merely demonstrates some realistic way of how the trigger and arguments are related in a text. The trigger and arguments with their roles are explicitly marked in the sentence, linking their appearances to their respective lists. For example:

  \[\text{The mobsters fired at Smith and his daughter.}\]

A spec can be conveniently encoded as an XML file. Figure 4.3 demonstrates such file, as it is provided to the system.

The extended problem setup relies on a spec for each target event type, and find similarities between argument candidates in text and the spec’s argument samples, and between the trigger-argument-candidate links in text and the corresponding links in the spec’s usage samples.

### 4.2 Method

We address the extended problem setup by extending our seed-based method (see 3.2). We next bring in detail the main modifications to the seed-based method that comprise the spec-based one.

#### 4.2.1 Labels

In the seed-based method, we only iterate through tokens, assigning each with a trigger label. In the extended method, however, we return to the full setting utilized by the fully-supervised system of Li et al. (2013) (see section 2.3.1). For
Figure 4.3: Event spec XML (partial). Triggers and arguments are marked in the usage samples in brackets, via identifying markers defined for each of them.
each scanned token, if it is classified as a trigger, we go through all of the sentence’s \( m \) available argument candidates \( \mathcal{E} = \{ e_k \}_{k=1}^m \) (entities, values and time-expressions).

Nevertheless, unlike the fully-supervised system but similar to the seed-based one, we cannot label the argument candidates with concrete event-specific roles, as future target event types (along with their argument roles) are not known during training. So with accordance to our seed-based method, also our argument candidates are labeled using a binary label set \{\top, \bot\}, where labeling \( e_k \) with \( \top \) means it is an argument, and labeling it with \( \bot \) means it is not.

To support this label set, we must consider each argument candidate multiple times, once for each relevant role \( r \) in the spec. Thus, we process each sentence \( x \) once for each event type \( t \in \mathcal{T} \), and for each token \( x_i \) that is labeled as a trigger \( (y_i = \top) \) we process each argument candidate \( e_k \) once for each role of the current event type \( r \in t_R \). This way, each candidate argument is considered as a potential argument of each possible trigger token in the sentence, and potentially as holding each possible role. This of course means that the same candidate argument can be an argument of multiple triggers (meaning that it participates in multiple events), which is definitely a possible and valid scenario (see example sentence (1) in section 2.2, page 8).

### 4.2.2 Assignment Scoring

Scoring an assignment works similarly to the way it does in the seed-based system (see section 3.2.2). A feature instance \( \phi \) is comprised of feature type \( f \in \mathcal{F} \) and label \( l \in \{\top, \bot\} \), where \( f \) yields a numerical value (specifically, 0 or 1) that is used as feature value \( f_{\phi} \). However, there are some differences in the spec-based method:

1. We use the full spec of event type \( t \), marked \( \tilde{\Psi}_t \), and not only trigger seed list \( \Psi_t \).

2. When using a feature type \( f \), we apply it not only to token \( x_i \) and spec \( \tilde{\Psi}_t \), but also to a specific argument candidate \( e_k \). Additionally, since \( f \) is always applied in a context of a specific role \( r \in t_R \), we consider only the parts
in the spec that are relevant for $r$, marked $\Psi^r_t$. Thus, we apply $f$ in the following manner: $f(x_i, e_k, \Psi^r_t)$.

4.2.3 Feature Types

The feature types we have in the seed-based system (section 3.2.3, page 28) are in fact trigger features. We now introduce new feature types that operate on arguments, or argument features. Both kinds of feature types are used in the spec-based system, as it handles both triggers and arguments.

Both kinds of features are disjunctive, or “any”-based, as mentioned for trigger features in section 3.2.3. This means that when a feature uses some list of values from the spec (e.g. argument samples), then if similarity is found for one value in the list, it is considered to have been found for the entire list. This, of course, can be expanded to more sophisticated weighted methods for working with a list of values.

Argument features internally divide to two categories: stand-alone argument features and linked argument features. Features are divided to these categories based on which kind of information they use, and how they extract this information from the spec. We next describe these two categories in detail, and list the different features we used in each category.

Note that since we are bringing negative results, many of the features have multiple variations that we tried during our experiments, with varying results. Therefore, we cannot bring one solid list of features that we used. We do, however, specify in detail the different variations that we examined, as ideas for future research.

4.2.3.1 Stand-Alone Argument Features

Stand-alone argument features consider only an argument candidate, much like trigger features consider only a token as a trigger candidate. For that, they use the argument samples portion of the spec, by evaluating each argument candidate against each argument sample of a certain role. These would typically use lexical knowledge. For instance, a stand-alone argument feature can check if an argument
candidate $e_k$ and an argument sample $\tilde{\psi}$ are WordNet synonyms:

$$\phi_f(e_k, \tilde{\psi}) = \text{Synonym}(\text{“robber”, “thief”})$$

The concrete stand-alone argument feature types we used are identical to the ones we used for triggers (see Table 3.3, page 29): Same Lemma, Synonym, Hypernym, Similarity Relations, all based on WordNet. This is because we are trying to assess the same kinds of similarity, also for arguments. We did examine some variations of those, like not using derivations, and using a different level of transitivity (e.g. level 3 is a hypernym of a hypernym of a hypernym).

### 4.2.3.2 Linked Argument Features

Linked argument features consider both the argument candidate $e_k$ and the trigger candidate $x_i$. This way they can address any kind of link between them, like the path connecting them in a syntactic dependency graph. The relevant spec portion would be the usage samples, which is utilized in the following fashion: Given a usage sample, we pick the trigger appearing in the sentence (marked “TTT” according to Figure 4.3), and an argument sample from the sentence that complies with the current role $r$ (using its relevant mark, for instance “ATR” for Attacker). Now we can compute some value over the link between the trigger and the argument, either over the surface form of the sentence (e.g. the number of tokens between them), or over some deeper analysis of the sentence (e.g. use a parser to build a dependency graph of the sentence, and extract the dependency path between the trigger and the argument). The same computation can then be performed to the input text sentence with regards to its current trigger and argument candidate. Finally, the two computed values, one over the input text and one over the spec, can be compared and yield some similarity result.

For example, we consider the input text sentence in Figure 4.4a and the spec usage example in Figure 4.4b. We apply a dependency parser to both, resulting in the given dependency graphs. During decoding, in the input text sentence $x$ we get to trigger candidate token $x_i = \text{“shot”}$, argument candidate $e_k = \text{“handgun”}$ and role $r = \text{Instrument}$. We then address the corresponding elements in the usage sample sentence $\tilde{\psi}$, getting trigger candidate token “stabbed”, and using the
Figure 4.4: Dependency graphs of sentences from text input and from the usage samples portion of the spec. Both graphs have the following dependency path marked in bold: $[T](\text{prep}(\text{pobj}[A]))$

**Instrument** role we get to argument “knife”. We compute the dependency path between the trigger and argument in both cases, and now, a feature can yield if they are equal or not:

$$
\phi_f(x_i, e_k, \tilde{\psi}) = \text{equal}(\text{path}(x, \text{“shot”}, \text{“handgun”}), \text{path}(\tilde{\psi}, \text{“stabbed”}, \text{“knife”})) = \text{equal}([T](\text{prep}(\text{pobj}[A])), [T](\text{prep}(\text{pobj}[A]))) = 1
$$

For the dependency path we use a parentheses notation, where a dependency (e.g. prep) has its subtree surrounded in parentheses. We mark the trigger’s location with $[T]$, and the argument’s location with $[A]$. We can clearly see in the example of Figure 4.4 that both dependency paths are equal, and therefore a linked argument feature that is testing for that can yield the positive value 1.

All concrete linked argument feature types we examined were variations of checking for dependency path equality, as illustrated in the previous example. Each sentence from the input texts and from the spec usage examples was analyzed with the EasyFirst dependency parser (Goldberg and Elhadad, 2010), and
<table>
<thead>
<tr>
<th>Param</th>
<th>Values</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| Root  | False, True | When True, the path also includes the dependency from the path’s root to its parent | \( F: [T]\{\text{dobj}[A]\} \)  
\( T: \text{ccomp}([T]\{\text{dobj}[A]\}) \) |
| Flat  | False, True | When True, some dependencies are converted to a more generic category, according to the Stanford dependencies manual. Specifically, this is done to categories \text{obj}, \text{mod} and \text{comp}. | \( F: [T]\{\text{dobj}[A]\} \)  
\( T: [T]\{\text{obj}[A]\} \) |
| Prep  | False, True | When True, the \text{prep} dependency is added the preposition itself. | \( F: [T]\{\text{prep}\{\text{pobj}[A]\}\} \)  
\( T: [T]\{\text{prep}_\text{with}\{\text{pobj}\{[A]\}\}\} \) |
| POS   | False, Generic, Specific | When non-False, adds a generic/specific part-of-speech to the path. | \( F: [T]\{\text{dobj}[A]\} \)  
\( G: [T]\{\text{dobj}[/\text{NNP}][A]\} \)  
\( S: [T]\{\text{dobj}[/\text{NNP}][A]\} \) |
| Up    | \( \infty, 1, 2, 3 \) | Limit the amount of dependency edges from the argument and up. Often we won’t get all the way to the trigger. | \( \infty: [T]\{\text{prep}\{\text{pobj}[A]\}\} \)  
\( I: \text{\{pobj\{[A]\}\}} \) |
| Both  | False, True | When True, we consider both path equality and some stand-alone argument feature over the argument. | \( F: \text{equal}(\text{path}_x, \text{path}_\psi) \)  
\( T: \text{equal}(\text{path}_x, \text{path}_\psi) \land \text{Synonym}(e_k, \text{arg}_\psi) \) |

Table 4.1: Parameters constructing different dependency-path-based linked argument features

all dependency paths were extracted from the resulting dependency graphs. A dependency graph is essentially a tree fragment, in which many times the root of the fragment is the trigger, and the argument is a single leaf. This is, however, not always the case, as sometimes the trigger and argument are two leaves of a tree fragment with some arbitrary root connecting two branches. For instance, in the sentence: “We go to war”, the dependency path between argument “We” and trigger “war” is \((\text{nsubj}[A])\{(\text{prep}(\text{pobj}[T]))\})\), where \([A]\) and \([T]\) are in two separate branches (the root is “go”, but is not mentioned in this notation, as the notation only contains dependencies leading up to the root, not the root itself).

Table 4.1 lists the different parameters we used in creating the different variations of features, where each variation was comprised of some single choice of value for each of the parameters. For instance, one dependency-based-feature could be: \( \langle \text{Root} = F, \text{Flat} = F, \text{Prep} = T, \text{POS} = G, \text{Up} = 2, \text{Both} = F \rangle \).

4.3 Evaluation

We bring forth the setting and results of our evaluation. This is while keeping in mind that our results, albeit not revolving around absolute zero, were low enough for us to deem them as negative and cease analysis of the spec-based approach at this point. Therefore, the evaluation is not at the same level of maturity and completeness as the final one used for the seed-based approach. We do advise future research, however, to make use of our proposed spec-based approach and leverage it to a complete and sound system, which will be evaluated and compare to previous results in full.

4.3.1 Setting

Our evaluation setting of the spec-based system is similar to the one we used for the seed-based system (see 3.3.1). We use the same documents with the same split (see Table 3.4), and use the same system parameters, such as beam size $K = 4$. We use the same approach of evaluating over a specific test event type, by training over a small pre-selected group of training and development event type. We also run the test 10 times for each of those test event types, randomly choosing the participating training event types and development event types for each run, to avoid bias of some specific choice.

Three main differences do exist between the spec-based evaluation setting and the seed-based one:

1. The obvious first difference is the fact that we didn’t use seed-lists but rather full specs. We built the specs in full for the used types (as required by the spec structure in Figure 4.2).

2. As our evaluation of the spec-based approach ceased before reaching full maturity, we used only several event types. These were the 7 types: Attack, Meet, Die, Marry, Divorce, Be-Born and Injure. These were alternated between training, development, and test event types, as our setting requires. These event types were selected as representatives of the entire set of 33 ACE event types, as they have versatile amounts of annotated triggers and
arguments in the test documents. For example, Attack has 95 annotated triggers and 122 annotated arguments in the test documents, whereas Marry has only 10 annotated triggers and 18 annotated arguments in these documents.

3. The third and final difference stems from the fact that unlike the seed-based system, we now needed to decide how to retrieve our argument candidates (which, as mentioned in section 2.3, is not in the scope of the fully-supervised system, and therefore also not in the scope of our system). We decided to retrieve our arguments from the gold standard, as Li et al. (2013) do for most of their published results. It is of course possible to re-run our experiments over the argument candidates provided by the external IE system, that Li et al. (2013) used for the other part of their results.

### 4.3.2 Results

We measure micro-average performance over our evaluation setting. We measure “Argument Role” performance, as termed by Li et al. (2013). This means that a positive match is counted only when an argument retrieved by the system matches a gold standard argument by satisfying 3 conditions: (1) They contain the exact same tokens in text; (2) They are both connected to the same trigger; (3) They both have the same role.

The micro-average results we measured are: Precision=60.7%, Recall=20.4%, and $F_1=28.7%$. For reference, the “Argument Role” results of Li et al. (2013) were: Precision=64.7%, Recall=44.4%, and $F_1=52.7%$. Note that these results are not comparable, as they were measured over different sets of event types.

The most prominent problem that arises from these results is the low recall. This indicates that our features cannot effectively capture many valid instances of arguments in text. There could be several reasons for that, such as:

1. WordNet contains mostly common nouns, yet arguments are often proper nouns, so they will have no relations to WordNet terms.

2. The dependency path between an argument and its trigger can be very versatile across different text sentences, even for the same role. The small number
of usage samples, along with the different variations of dependency-based features we have, does not seem to cover enough cases.

These caveats can be potentially alleviated, or even potentially resolved, with more extensive research of the topic. Specifically, we encourage the use of more knowledge resources and external tools, that can cover more semantic and syntactic variability.
Chapter 5

Conclusions and Future Work

5.1 Conclusions

As performance was vastly different between trigger labeling and full event extraction, our conclusions are divided as well.

For trigger labeling, we showed the strength of the combination of information embedded in annotation guidelines of event extraction datasets and external lexical resources. These two elements encapsulate a sufficient amount of quality knowledge to allow high-performance extraction of target event types. Such high-performance usually comes with the overbearing cost of vast manual annotation, which can often make the entire task infeasible. This could happen, for instance, when the required target event does not appear in enough available texts, e.g. when it describes some new phenomenon in the world that has yet to be fully explored. Another case could be low-resource languages, where the amount of available texts is low to begin with. But even in the most common scenarios of event extraction, the cost of annotation is always a painful part of the process. As we match performance of a state-of-the-art fully-supervised system over the ACE-2005 benchmark (and even surpass it), we offer our approach as an appealing way of alleviating this pain while still preserving classification quality. Who could say no to that?

As for the full event extraction, we believe we carved the beginning of the right path. The performance doesn’t quite reflect this yet, but we believe that
by combining argument examples with argument connections to their triggers, a system will be able to generalize at least some of the target roles. The inherent difficulty of this task made us stop after some amount of attempts, but the potential is still there. We should note that even if future research achieves performance that is higher than ours but still below state-of-the-art as obtained by supervised systems, this may still be an effective contribution - this is due to the fact that the major appeal of this approach is eliminating most of the manual annotation and allowing any future target event types. So even with some performance hit, the system could be relevant for many scenarios (as mentioned earlier).

5.2 Future Work

There are several appealing directions for potential future work, each extending our work in a different manner. We list some such ideas. Note that the ideas presented here apply both for triggers and for arguments, as they are relevant to the more generic parts of the system.

**Non-boolean feature values.** Our method defines that features yield a numeric value (used as feature value $f_\phi$), and that for simplicity, we used only boolean values 0 and 1 (see section 3.2.2). However, a more rich feature-value-space can be used, that represents the degree of similarity in a much more fine-grained fashion.

One example for that could be to give a different score according to the level of transitivity used for matching. For instance, if token $x_i$ has seed $\psi$ as a direct hypernym, that would yield a high score, whereas if $x_i$ has seed $\psi$ 3 levels up in the hypernym hierarchy (a hypernym of a hypernym of a hypernym), that would yield a lower score. Even though technically in both cases $x_i$ would equally be a subtype of $\psi$, we can expect some semantic drift and larger ambiguity the more two terms are far apart from each other in the WordNet graph.

Another example comes from the fact that currently WordNet returns $False$ for two terms if it determines that the queried relation doesn’t hold between the terms, but also when WordNet cannot provide an answer to the query. This could happen when at least one of the terms does not appear in WordNet at all. In this case, we would expect it to return a “softer” $False$ value, as it cannot positively
determine that the terms are unrelated. For these cases, a different value can be returned, logically marked “Undetermined”, having some appropriate numeric value between 0 and 1.

**More knowledge resources.** In this work, we only used WordNet as a lexical similarity resource (for triggers and argument samples), and EasyFirst dependency parser as a syntactic resource (for usage samples). The NLP world is full of additional valuable resources for both of these resource types. For lexical resources we could use Nomlex nominalizations lexicon (Macleod et al., 1998), Brown clusters (Brown et al., 1992; Miller et al., 2004; Sun et al., 2011), paraphrase databases (like in (Ganitkevitch et al., 2013)), and others. For syntactic-related resources, we could use semantic-role labeling, such as in (Punyakanok et al., 2008). The sky is the limit.
Bibliography


