

# Preposition Sense Disambiguation - Dokumentation

Dirk Neuhäuser  
7094369

Tim Rosenkranz  
6929884

Tobias Marzell  
123456789

22. September 2020

## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Datenbeschaffung und -bereinigung</b>	<b>3</b>
<b>3</b>	<b>Huggingface Transfores mit Bert</b>	<b>4</b>
3.1	Trainer . . . . .	4
3.2	Tagger und Einbindung . . . . .	5
<b>4</b>	<b>FlairNLP</b>	<b>6</b>
4.1	Trainer . . . . .	6
4.1.1	Aufbau Korpus . . . . .	6
4.1.2	Aufbau Classifier . . . . .	6
4.1.3	Embeddings . . . . .	7
4.2	Anwendung . . . . .	7
4.2.1	Trainieren . . . . .	7
4.2.2	Satz predicten . . . . .	8
4.3	Integration TextImager . . . . .	8
<b>5</b>	<b>Semi-supervised</b>	<b>9</b>
5.1	Daten . . . . .	9
5.1.1	Datenaufbereitung . . . . .	9
5.1.2	Training . . . . .	9
5.1.3	Ergebnisse . . . . .	10

## 1 Einleitung

In dem Praktikum *Text2Sene* geht es darum, aus Textbeschreibungen Szenen zu erstellen. Dabei wurde die Arbeit unterteilt. Wir beschäftigen uns mit der Thematik der *preposition sense disambiguation* (Sinneszuordnung und -erkennung von Präpositionen). Für diese Aufgabe haben wir verschiedene *State-of-the-Art-Verfahren* begutachtet. Schlussendlich haben wir uns dafür entschieden,

einerseits mit der FlairNLP library(Tim) und der Huggingface-transformers library(Dirk) jeweils ein supervised Modell umzusetzen und andererseits einen *Semi-supervised* Ansatz(Tobias) zu verfolgen.

## 2 Datenbeschaffung und -bereinigung

TIM UND DIRK

Die Aufgabe zur Disambiguieren von Präpositionen war bereits 2007 eine Aufgabe in der SemEval<sup>1</sup>. In mehreren Publikationen bezüglich preposition sense disambiguation wurde dieser Datensatz als Benchmark verwendet, obwohl er im Internet nur schwierig zu finden ist. Deshalb ist dieser Abgabe auch der originale Datensatz mit angefügt (TPPCorpora.zip). Der Datensatz liegt in mehreren xml Dateien vor. Für Jede Präposition gibt es einerseits eine xml Datei mit Trainingsätzen und eine xml Datei mit den verschiedenen Sinn-Bedeutungen der Präposition.

Hier ein Beispiel der Präposition 'with'. Das `<head>` `</head>` umschließt dabei immer die Präposition die disambiguiert werden soll:

```
<instance id="with.p.fn.338359" docsrc="FN">
  <answer instance="with.p.fn.338359" senseid="7(5)" />
  <context>
    She nodded <head>with</head> enthusiasm .
  </context>
</instance>
```

Der Sinn hinter diesem spezifischen 'with' ist mit der Senseid 7(5) deklariert worden. In der Definition xml für diese Präposition steht dazu folgendes:

```
<sense id="6">
  <definition>indicating the manner or circumstances (
    but not cause or motivation) of something (e.g.,
    fix with precision)</definition>
  <majorcluster> MANNER </majorcluster>
  <pprojmap type="equivalent" targetid="7(5)" />
</sense>
```

Allerdings ist der Datensatz teilweise etwas inkonsistent und daher mussten die Daten zunächst um NA Einträge, fehlende Sinne, oder fehlende Sinn-Definitionen bereinigt werden. Das Resultat haben wir in einer tsv-Datei zusammengefasst und enthielt insgesamt 16397 Sätze mit Sinneszurordnung. Hier zum Beispiel ein kleiner Ausschnitt der Datei:

id	sentence	labelid	definition
8	She knelt <head>on</head> the cold stone floor and carefully placed some coals on the dying embers in the grate .	13	physically in contact with and supported by (a surface) (e.g., the book on the table)
9	The eleventh commandment : Thou shalt not lean <head>on</head> the left elbow, or else.	3	indicating the part(s) of the body supporting the rest of the body (e.g., stood on his feet)

Die tsv Datei der bereinigten Sätze ist ebenfalls der Abgabe beigelegt (training\_data.tsv).

<sup>1</sup>SemEval Aufgaben beschäftigen sich mit der Word Senses und Beziehungen von Wörtern in Sätzen

### 3 Huggingface Transformes mit Bert

DIRK

Die Huggingface transformers library stellen einheitliche und allgemeine state-of-the-Art Architekturen bereit. Die unterstützen Modelle sind äußerst gut vortrainiert und gehören im NLU Bereich zu den besten. Z.B. bert, ein von google trainiertes Modell, knackt gleich in mehreren Bereichen die state-of-the-Art. Zum Disambiguieren haben wir uns deshalb für bert entschieden. Da die Aufgabe darin besteht einer Präposition **eine** der mehr als 200 Sinnklassen zuzuordnen, wurde BertForSequenceClassification gewählt. Ein Modell, welches man nur noch mitgeben muss wieviele Klassen es und man erhält dirket vortrainiertes Modell samt gewichten und bereits korrekter Layer - Architektur. <sup>2 3</sup>

#### 3.1 Trainer

Zum Trainieren des Taggers wurde ein Skript torch\_trainer.py entwickelt, welches auf PyTorch aufbaut.

**Das Skript hat insgesamt 3 Phasen:**

1. Einlesen und Tokenisieren der Daten
2. Modell initialisieren und konfigurieren
3. Fine-tunen von Bert

**Good to know:**

- Im ersten Schritt, wurde ein 90:10 Trainings-Validierungs Split durchgeführt.
- Der Tokenizierer von der Huggingface transformer library ist. Tokenizer müssen auch trainiert werden und Huggingface liefert für die Bert Modelle die, die Englische Sprache bedienen wollen, schon vortrainierte und sehr gute Tokenizer. Die Tokenizer wandeln Sätze dann in inputIds und attentionMasks, eine Repräsentation der Wörter in Zahlen.
- Trainingsschleife ist eine Standard-Pytorch Implementierung<sup>4</sup>
- Hyperparameter-Optimierung - Das google-Research Team um bert empfiehlt<sup>5</sup>:
  - 4 epochen
  - Adam-Optimizer
  - batch sizes mit den Werten 8, 16, 32, 64 oder 128
  - learning rates mit den Werten 3e-4, 1e-4, 5e-5, 3e-5

---

<sup>2</sup>Offizielle Dokumentation: <https://huggingface.co/transformers/>

<sup>3</sup>Offizielles Repository: <https://github.com/huggingface/transformers>

<sup>4</sup>Orientierung an: <https://mccormickml.com/2019/07/22/BERT-fine-tuning/>

<sup>5</sup><https://github.com/google-research/bert>

Das beste Ergebnis wurde mit einer batch-size von 16 und einer learning-rate von 1e-4 erzielt. Die Val-Accuracy konnte mit diesen Parametern insgesamt 0.9084 erreichen.

Zum Selber trainieren müssen die Packages torch, transformers und wandb installiert sein und die **training\_data.tsv** in einem Ordner namens data eingefügt werden. Da wandb zum loggen und displayen der metrics verwendet wird, muss vor dem Start noch am Anfang des Skriptes ein Api-key angegeben werden (falls nicht vorhanden noch ein Account vorher erstellt werden). Genauso müssen Werte für batch-size und learning-rate zu Beginn des Skriptes gesetzt werden. Als default Werte wurde eine batch-size von 16 und eine learning-rate von 1e-4 gewählt, da diese in der Hyperparameter-Optimierung am Besten abgeschnitten hatten. Am Ende des Trainings erhält man einen Ordner namens model\_save, welche die Gewichte und die Config in Pytorch Format enthält. Möchte man zu einem spätern Zeitpunkt nochmal retrainen, kann in der torch\_trainer.py Datei das Modell statt von den pretrained Gewichten, von unseren Gewichten geladen werden:

Also der folgende Teil:

```
model = BertForSequenceClassification.  
    from_pretrained(  
        "bert-base-uncased",  
        num_labels = len(data_train.training_label.  
            value_counts()),  
        output_attentions = False,  
        output_hidden_states = False,  
    )
```

wird zu:

```
model = BertForSequenceClassification.  
    from_pretrained("model_save")
```

### 3.2 Tagger und Einbindung

Der beigefügte tagger ist sehr einfach aufgebaut. In der **init-Methode** wird das Modell aus dem Ordner model\_save geladen und in der **tag-Methode** wird für ein mitgegebener Satz eine LabelId returned, die das Modell predicted. In der Einbindung in den **Text-Imager** wird für jedes Preposition-Token ein **Word-Sense** mit dieser LabelId beigefügt. Zum benutzen fehlt noch das trainierte Model. Dazu legt man in resources/model\_save die .bin und die .config Datei ab. Der Abgabe wurde ebenfalls eine Datei Names\_definitions.tsv beigefügt, welche eine Map der LabelIds auf ihre Definitionen enthält (falls Interesse daran besteht).

## 4 FlairNLP

TIM

FlairNLP [Akbik, Blythe und Vollgraf 2018]<sup>6</sup> ist ein Python-Framework, das speziell für NLP-Aufgaben konzipiert ist und auf **PyTorch** basiert. Für unsere Aufgabe nutzen wir einen *text classifier*, der wie der Name sagt, eine Eingabe klassifiziert und dadurch die Präposition dem zugehörigen Sinn zuordnet. Für die Verwendung von Flair ist **PyTorch** vorausgesetzt. Zudem wird **hyperopt** benötigt, sofern eine Optimierung<sup>7</sup> vorgenommen werden soll. Zum Ausführen des Trainings-Skripts wird mindestens **Flair Version 0.6** vorausgesetzt, da bei früheren Versionen die Embeddings nicht mehr herunter geladen werden können. Diese Version ist zudem auch eine generelle Empfehlung.

### 4.1 Trainer

Das Trainieren des Classifiers wird mit dem Skript `flair_model.py` vollzogen. Es bietet die Möglichkeiten ein Modell mit wählbaren Hyperparametern zu trainieren oder zu optimieren. Standardmäßig werden bereits optimierte Werte verwendet.

#### 4.1.1 Aufbau Korpus

Der Classifier ist der Hauptbestandteil dieses Flair-Projekts. Für das Training dieses Classifiers ist aber ein Korpus notwendig, in dem die Trainingsdaten verarbeitet werden. Dies geschieht in der Methode `_create_corpus`. Die column name map (`col_name_map`) wird hier angegeben und enthält die Information, wo in der Datei der Daten die label und wo der Text steht. Der Korpus wird dann mit der column name map, den Trainingsdaten und einem Tokenizer erstellt und in der Klassenvariable `__corpus` gespeichert. Die Trainingsdaten müssen dabei im csv-Format abgespeichert sein und ggf. in 80% Train, 10% Dev und 10% Test aufgeteilt. Sofern keine Test- und Dev-Datei besteht, wird aus den Trainingsdaten ein Anteil genommen, um Dev- und Test-Datensätze zu erstellen (Flair standard). Für diese Aufgabe steht zudem das Skript `flair_prepare.py` zur Verfügung. Hierbei kann die Methode `version1` genutzt werden, um die Daten direkt aus den xml-Dateien zu lesen oder die Methode `version2`, um die Daten aus der aufbereiteten tsv-Datei zu entnehmen. Zusätzlich werden die Sinnes-IDs mit der flair-Standardmarkierung `__label__` markiert

#### 4.1.2 Aufbau Classifier

Existiert ein Korpus, kann ein Classifier (in der Methode `_create_classifier`) erstellt werden. Zuerst wird dazu ein dictionary der label auf Basis des Korpus erstellt. Anschließend werden die gewünschten **Embeddings** erstellt. Danach kann mit

<sup>6</sup>Github: <https://github.com/flairNLP/flair>

<sup>7</sup>Siehe: [Flair Tutorial 8](#).

diesen beiden Daten der Classifier erstellt werden. Hierbei geben wir auch mit an, dass ein Satz keine zwei Klassen besitzen kann (*multi\_label = False*).

### 4.1.3 Embeddings

FlairNLP liefert einige Embeddings. Mit den GloVe-WordEmbeddings zusammen mit selbstlernenden OneHotEmbeddings lassen sich gute Ergebnisse erzielen. Zum kombinieren der Embeddings bieten sich die *DocumentPoolEmbeddings* Akbik, Bergmann und Vollgraf 2019 an.<sup>8</sup>

## 4.2 Anwendung

Die Anwendungen Trainieren und Predicten sind in zwei Skripte getrennt. Letzteres ist in das **TextImager**-Projekt eingegliedert.

### 4.2.1 Trainieren

Das Trainieren geschieht mit dem Skript `model_flair.py`<sup>9</sup> und den benötigten Trainingsdaten. Für das trainieren muss die Methode *train()* angewandt werden. Hierbei können vier Parameter übergeben werden, das Verzeichnis, in dem die Daten liegen (standardmäßig *data*), die mini batch size, learning rate und die Anzahl der Epochen. Diese sind mit Standardwerten versehen, die gute Ergebnisse erzielen.

Bevor das Training beginnt, wird wenn möglich ein bereits existierender Classifier geladen, der sich in dem Verzeichnis befindet, das beim Erstellen des Objektes angegeben wurde. Falls keiner existiert, wird ein neuer erstellt. Darüber hinaus wird immer ein Korpus erstellt. Dabei muss mindestens eine datei mit dem Namen *train.csv* im Verzeichnis liegen, optional auch eine Datei *dev.csv* und *test.csv* - andere Namen können ohne weiteres **nicht** gewählt werden.

Sofern kein anderes verzeichnis angegeben wurde, wird das trainierte Model, sowie log-Files in dem Verzeichnis *resources* gespeichert. Das finale Model trägt den Namen „*final-model.pt*“.

### Konfiguration und Ergebnisse

Das Flair Model in dem Skript *flair\_model.py* hat folgende Werte erreicht:

- F-score (micro) 0.9549
- F-score (macro) 0.635
- Accuracy 0.9549

Dabei können die Werte u.U. aber auch niedriger ausfallen, dies sind Maximalwerte.

Folgende Konfiguration ist dabei verwendet:<sup>10</sup>

<sup>8</sup>Siehe: [Flair Tutorial 3](#) und [Flair Tutorial 5](#) für mehr Infos.

<sup>9</sup>Das Skript *model\_flair.py* enthält die Einstellungen für ein gutes Ergebnis. Das Skript *flair\_model.alt.py* besitzt zudem noch eine alternative Konfiguration, die jedoch schlechter ist.

<sup>10</sup>In der Optimierung können mehr werte als hier angegeben konfiguriert werden. Dies liegt daran, dass die Optimierung mit DocumentRNNEmbdings arbeitet, wir mit DocumentPoolEmbeddings.

- mini batch size = 16
- learning rate = 0.1
- patience=5 [kann ggf. variiert werden]
- Embeddings = GloVe + OneHot; DocumentPoolEmbeddings (Siehe auch: [Embeddings](#))
- Tokenizer: SpaceTokenizer

#### 4.2.2 Satz predicten

Das Predicten wird in dem Skript `flair_disambiguation.py` vollzogen. Um einen Satz zu predicten, muss lediglich ein *String* in die Methode `predict()` übergeben werden. Dabei muss die Präposition, die klassifiziert werden soll mit den html-Tags `<head>` und `<\head>` umschlossen sein. Es kann nur eine Präposition predictet werden. Bei Missachtung können fehlerhafte Ergebnisse ausgegeben werden.

In diesem Skript muss beim erstellen des Objekts der Pfad angegeben werden, in dem das Classifier-Model liegt. Ein dort befindlicher Classifier mit Namen **best-model.pt** wird dann geladen. Alternativ kann auch die Methode `_load_classifier()` verwendet werden. Sofern bei einem Aufruf der `predict`-Methode kein Classifier geladen werden kann, wird abgebrochen. Es wird die Sinnes-ID ausgegeben, keine Definition. Sofern an der Definition für die Sinnes-IDs besteht, kann die Map der LabelIDs genutzt werden, die mit Huggingface bereitgestellt wird.

### 4.3 Integration TextImager

In den TextImager<sup>11</sup> wurde nur das Skript zum predicten, `flair_disambiguation.py`, integriert. Das Trainieren muss mit dem entsprechenden Python-Skript erfolgen.

Die (Java) Klasse **FlairDisambiguation** stellt dabei das Interface zum korrespondierenden Python-Skript dar. Es ist so aufgebaut, dass es mit einem *AggregateBuilder* ausgeführt werden kann. Für diesen sind die Methoden `initialize` und `process` wichtig. In ersterer wird eine Python-Umgebung initialisiert und vorbereitet, sodass in der Prozessmethode lediglich predicted werden muss. Dabei wird jede Präposition einzeln predicted und die Sinnes-ID in das JCas hinter die Präposition eingefügt.

Mit der Klasse **TestFlairDisambiguation** kann ein Test vorgenommen werden.

Damit das Klassifizieren funktioniert, muss in dem Ordner *resources* der Maven-Struktur das trainierte Model abgelegt und im Code der Pfad ggf. angepasst werden. Weiterhin ist es notwendig, vor das Flair Model einen **Part-Of-Speech** Tagger vorzuschalten, damit die Präpositionen korrekt mit den html-Tags versehen werden können.

---

<sup>11</sup>Siehe: <https://github.com/texttechnologylab/textimager-uima>



## 5 Semi-supervised

TOBIAS

### 5.1 Daten

Die Daten die für das Training verwendet wurden befinden sich auf der Seite: <http://www.statmt.org/europarl/>. Ich habe den Deutsch-Englisch Korpus verwendet, prinzipiell sollte aber jeder funktionieren.

#### 5.1.1 Datenaufbereitung

Um die Daten in ein trainierbares Format zu bekommen, wird in dem originalen Paper (<https://arxiv.org/abs/1611.08813>) der **CDEC Word-Aligner** benutzt. Zunächst sollte man die Sätze der jeweiligen Sprachen jeweils in eine eigene Zeile aufteilen. Dazu kann das Script `split-sentences.perl` von den Europarl tools verwendet werden. Danach sollten die einzelnen Sätze tokenized werden. Dazu kann der `tokenizer.perl` ebenfalls aus den Europarl tools verwendet werden. Danach sollten die Skripts aus dem Ordner "Corpus" von CDEC verwendet werden. Die Reihenfolge der Skripts ist prinzipiell egal. Nach der Ausführung der einzelnen Skripts kann es passieren, dass einige der Sätze einer Sprache in der anderen nicht vorkommen. Um das Problem zu beheben, habe ich mich dafür entschieden ein Python Skript zu schreiben, dass alle Sätze entfernt, die nur in einer Sprache vorkommen. Das Skript heißt `clean.py`. Das Skript liest aus dem `stdin` und schreibt in den `stdout`. Demnach bietet es sich an mit Bash Redirects einen File als Input zu redirecten und in einen File zu speichern. Beispielsweise über den Aufruf `python3 clean.py < input.txt > output.txt`.

Danach sollten die Daten in folgendem Format sein:

tokenized english sentence . ||| dazu passender tokenizierter deutscher satz .

Mit diesem Format kann der Word-aligner von CDEC ausgeführt werden. Ich habe dazu das `fast-align` Programm benutzt.

Mit dem resultierenden File kann nun das von mir geschriebene Python Skript `preparedata.py` ausgeführt werden. Dazu muss aber in dem Skript selbst der Pfad zu den Daten angepasst werden. Der erste geöffnete File sollte ein File sein in dem eine einzeilige, headerlose csv Datei sein, in der pro Zeile jeweils eine englische Preposition steht, der zweite geöffnete File muss der File zu den parallelen Textdaten sein und der dritte File muss der File zu den Alignment Daten sein. Nach der Ausführung des Skripts befindet sich nun der File `training.data` in dem Ornder in dem das Skript ausgeführt wurde. Die Daten sind nun so aufgebaut, dass die Tokens der linken Sprache ohne Präposition stehen, gefolgt von einem Tab Symbol, gefolgt von der Präposition in der anderen Sprache. Mit diesen Daten sind die Trainingsskripts auszuführen.

#### 5.1.2 Training

Beigefügt sind die Skripts `train.py`, `transferbert.py` und `gru.py`. Zum Training empfiehlt es sich ein **Virtualenv** anzulegen und die `requirements.txt` zu installieren.

### **5.1.3 Ergebnisse**

Die besten Ergebnisse erzielte gru.py.

Nach 97 Epochs besteht eine (Validation) Accuracy von (38,8%) 90,7%.

## Literatur

- Akbik, Alan, Tanja Bergmann und Roland Vollgraf (2019). “Pooled Contextualized Embeddings for Named Entity Recognition”. In: *NAACL 2019, 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. S. 724–728.
- Akbik, Alan, Duncan Blythe und Roland Vollgraf (2018). “Contextual String Embeddings for Sequence Labeling”. In: *COLING 2018, 27th International Conference on Computational Linguistics*. S. 1638–1649.