Preposition Sense Disambiguation - Dokumentation

Dirk Neuhäuser 7094369 | Tim XYZ 123456789 | Tobias XYZ 123456789

September 6, 2020

Contents

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 Aufage zur Disambigueren von Prepositionen war bereits 2007 eine Aufgabe in der SemEval[1]. 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 Datein vor. Für Jede Preposition 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":

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

Der Sinn hinter diesem spezifischen "with" ist mit der Senseid 7(5) deklariert worden. In der Definition xml 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:

[1]SemEval Aufgaben beschäftigen sich mit der Word Senses und Beziehungen von Wörten in Sätzen

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 Säte ist ebenfalls der Abgabe beigefügt (training_data.tsv).

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 vortrainertes Modell samt gewichten und bereits korrekter Layer - Archtiektur. [2] [3]

3.1 Trainer

Zum Trainieren des Taggers wurde ein Skript hftrainer.py entwickelt, welches auf PyTorch aufbaut.

Das Skript hat insgesamt 3 Phasen:

1. Einlesen und Tokenisieren der Daten
2. Modell initiailisieren und configurieren
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 trainieret 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.

[2] Offiziele Dokumentation: https://huggingface.co/transformers/
[3] Offizielles Repository: https://github.com/huggingface/transformers

- Trainingsschleife ist eine Standard-Pytorch Implementierung[4]
- Hyperparameter-Optimierung - Das google-Research Team um bert empfiehlt[5]:
  - 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

  Das beste Ergebnis wurde mit einer batch-size von XYZ und einer learning-rate von XYZ erzielt. Die Validiäts-Accurcy konnte mit diesen Parametern insgesamgt XYZ 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 **wandb login** im Terminal aufgerufen werden (falls nicht vorhanden noch ein Account vorher erstellt werden). Als default Werte wurde eine batch-size von XYZ und eine learning-rate von XYZ gewählt, da diese in der Hyperparameter-Optimierung am Besten abgeschnitten hatten. Am Ende des Trainings erhält man einen einen Ordner names 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 beigefügter hftagger.py 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 **WordSense** mit dieser LabelId beigefügt. Der Abgabe wurde ebenfalls eine Datei Names definitions.tsv beigefügt, welche eine Map der LabelIds auf ihre Definitionen enthält (falls Interesse daran besteht).

[4]Orientierung an: https://mccormickml.com/2019/07/22/BERT-fine-tuning/
[5]https://github.com/google-research/bert

4 FlairNLP

TIM

FlairNLP ist ein Framework, das speziell für NLP-Aufgaben konzipiert ist. Für unsere Aufgabe nutzen wir einen *text classifier*, der wie der Name sagt, eine Eingabe klassifiziert und dadurch die Präposition dem zugehörigen Sinn zuordnet.

4.1 Vorgehen [Training]

Damit einer Präposition ein Sinn zugeordnet werden kann, passieren einige Dinge. Der Hauptteil des Projekts besteht aus einer NLP-KI mit dem Flairframework.
Damit eine Eingabe verarbeitet werden kann, muss diese zunächst vorbereitet werden. Dazu werden die Daten aus den xml-Dateien gelsen und in einer csv-Datei mit dem entsprechenden Format für Flair abgespeichert. Hierbei wird das standard-Format (__label__<label>) verwendet. Die Trainingsdaten werden in drei Dateien aufgeteilt, dabei werden 80% Training, 10% Dev und weitere 10% Test zugeschrieben.
Nachdem die Daten im csv-Format vorliegen, wird daraus ein Korpus erstellt. Dazu wird ein *Dictionary* und die entsprechenden Embeddings erstellt. Anschließend wird der *Classifier* erstellt und das Training beginnt.[6]

4.2 Anwendung

4.2.1 Trainieren

4.2.2 Satz predicten

Um einen Satz zu predicten, ist nicht viel erforderlich. Beim Aufrufen muss lediglich eine Liste von *Strings* übergeben werden. Wurde beim erstellen des Objektes angegeben, dass nicht tokenisiert werden soll, werden die übergebenen Sätze nur mit einem *SpaceTokenizer* tokenisiert. Die übergebenen Sätze dürfen zudem noch nicht markiert sein - die **Markierung der Präpositionen wird automatisch** mit dem *Sequence Tagger* (↑) **vollzogen**.
Bevor predictet wird, wird zuerst geprüft, ob ein *classifier* vorhanden ist. Wenn nicht, wird wenn möglich einer importiert, ansonsten ein neuer erstellt - jedoch wird **immer** erst versucht, einen existierenden zu importieren. Hierbei ist wichtig, dass das bei der Erstellung des Objekts angegebene Verzeichnis als Quelle für Imports und Exports herangezogen wird.

4.3 Sequence Tagger - Präpositionen markieren

Der *Sequence Tagger* ist notwenidg, um in einem Satz die Präposition zu markieren, die klassifiziert werden soll, v.a. in Sätzen, in denen mehrere vorhanden sind.
Hierbei ist zu beachten, dass für eine einfache Anwendung des Flair Projekts zum predicten der Präposition in einem Satz das Verstehen und / oder Anwenden dieses Taggers **nicht** notwendig ist.

[6] Für eine genauere Erklärung bzgl. des Classifiers, des Korpus und des Dictionary verweisen wir auf die offizielle FlairNLP-Dokumentation.

4.3.1 Anwendung

Der *Sequence Tagger* ist ganz simpel zu benutzen. Zunächst muss wie gewöhnlich ein Objekt der Klassen erzeugt werden. Anschließend ist mit der Methode set_input() die Eingabe zu setzen. Die Eingabe muss eine Liste von *Strings* sein. Ist dies getan, kann mit do_tagging() das taggen gestartet werden. Diese Methode gibt dann eine Liste mit den reultierenden *Strings* zurück. Hierbei ist zu beachten, dass bei Eingabe eines Satzes mit zwei Präpositionen **zwei** Sätze zurückgegeben werden!

4.3.2 Funktionsweise

FlairNLP bietet mehrere bereits vortrainierte *Sequence Tagger*. Wir nutzen für unser Projekt den *Part-of-Speech Tagger*. Mit diesem wird der Satz zunächst predictet, wodurch jedem Wort der entsprechende Tag zugeordnet wird. Da wir uns aber nur für Präpositionen interessieren, löschen wir alle anderen Tags. Zeitgleich werden die Präpositionen aus dem Satz extrahiert, um sie später gezielt wieder einzusetzen und dabei jede Preposition in einem individuellen Satz markieren zu können.

4.3.3 Programm

Der *(Sequence) Tagger* ist in dem script *model_flair.py* enthalten.

- __init__: In der Initialisierungs-Methode Wird lediglich der zuvor beschriebene Tagger von Flair geladen.
- set_input: Diese Methode dient dazu, eine Liste an *Strings* zu übergeben, die getaggt werden sollen.
- do_tagging: Diese Methode taggt die zuvor in der set_input Methode übergeben Sätze. Rückgabeparameter ist eine Liste an *Strings*. Diese Liste kann u.U. größer sein, als die Liste der Eingaben.

4.4 Projektaufbau

Das Flair-Projekt umfasst hauptsächlich zwei Dateien:

- model_flair.py - Diese Datei enthält die Basisklasse(n) für das Projekt. Dabei sind nur die beiden Methoden *train* und *predict* der Klasse *Base-Model* für den einfachen Gebrauch notwendig. Erstere vollzieht das Training mit den der Methode angegebenen Daten (Übergabeparameter ist ein Verzeichnis) und zweitere predictet eine Liste an Sätzen (*Strings*), die übergeben werden.
- flair_test.py - Dieses script dient dazu, einen simplen test des Flair-Models durchzuführen. Es nimmt *command line arguments* entgegen, um zwischen Training und Testen, sowie ob ein Tokenizer verwendet werden soll, zu entscheiden. Das erste Argument muss dabei ein **boolean** sein und bezeichnet die Wahl für (Argument = *False*) oder gegen (Argument = *True*) einen Tokenizer. Das zweite Argument ist hingegen die Wahl für Training oder Test. Um ein Training zu starten muss "train" (*String*) eingegeben werden; Jede andere Eingabe wählt das Testen.

4.4.1 Aufbau des Classifiers und 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.

Korpus Bevor ein Classifier erstellt werden kann, muss zuerst ein Korpus erstellt werden. Dies geschieht in der Methode *_create_corpus*.

```
col_name_map = {0: "label", 1: "text"}
```

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.

```
if (not(self.use_tokenizer)):
    tokenizer = SpaceTokenizer()
else:
    tokenizer = SegtokTokenizer()
```

Als Tokenizer werden entweder der flair Tokenizer SegTok oder ein SpaceTokenizer verwendet, je nachdem ob angegeben ist, dass ein Tokenizer verwendet werden soll, oder nicht. Die Tokenizer können auch angepasst werden.

```
self.__corpus: Corpus = CSVClassificationCorpus
   (data_folder=data_dir,
    column_name_map=col_name_map,
    tokenizer=tokenizer)
```

Der Korpus wird dann mit der colun name map, den Trainingsdaten und dem Tokenizer erstellt und in der Klassenvariable *__corpus* gespeichert.

Classifier Existiert ein Korpus, kann ein Classifier erstellt werden.

```
label_dict = self.__corpus.
   make_label_dictionary()
```

Zuerst wird dazu ein dictionary der label erstellt.

```
word_embeddings = [WordEmbeddings('glove'),
   FlairEmbeddings('news-forward-fast'),
   FlairEmbeddings('news-backward-fast')]

document_embeddings = DocumentLSTMEmbeddings(
   word_embeddings, hidden_size=512,
   reproject_words=True,
   reproject_words_dimension=256)
```

Anschließend wenden wir Embeddings an. **WIP**

```
self.__classifier = TextClassifier(
   document_embeddings, label_dictionary=
   label_dict, multi_label=False)
```

Nachdem die Embeddings angewandt wurden, wird der classifier erstellt. **WIP WIP [old description]**
Nachdem der Korpus erzeugt wurde, werden die Embeddings erzeugt. Hierbei können mehrere Embeddings durch sog. *pool-embeddings* zusammen genutzt werden. Wir haben uns (hier) für OneHotEmbeddings und WordEmbeddings (Typ *glove)* entschieden[7].
Anschließend erstellen wir den Trainer auf Basis des zuvor erzeugten oder geladenen Classifier und Korpus und beginnen das Training. Hierbei können verschiedene Einstellungen vorgenommen werden. Zunächst wird das *output-directory* angegeben., welches wir mit *resources* angegeben haben. In diesem Verzeichnis werden dann logs und die Models abgespeichert. Die *learning rate* haben wir auf dem Sdandartwert belassen, ebenso die beiden weiteren Parameter. Die *patience* kann variabel angepasst werden. Sie verursacht, dass das Training bei zu vielen Epochen ohne Verbesserung **hintereinander** die Lernrate verringert oder das Training abgebrochen wird. Je höher die patience, desto mehr Epochen ohne Verbesserung können vorkommen. Der letzte Parameter ist die maximale Anzahl an Epochen. Wir haben einige Tausend Epochen trainiert. Dieses Training kann aber auch dauern; Auswirkungen auf die tRainingszeit haben u.a. auch die verwedeten Embeddings.

4.4.2 Aufbau des Trainings

4.4.3 Aufbau des Predictors

Der Predictor dient dazu, die Fertige KI anzuwenden. Um einen Satz korrekt zu predicten, muss in diesem die Präposition markiert werden. Diese Markierung ist ein html-Tag, <head>, bzw. <\head> und wird mit Hilfe des zuvor beschriebenen Taggers eingefügt. Diese Sätze werden dann mit der vom *flair-classifier* bereitgestellten Methode predictet.

[7] In anderen Dateien haben wir auch andere Embeddings getestet, aber mit diesen die besten Ergebnisse erzielt.

5 Semi-supervised

TOBIAS

Tobias war zu faul und hat nichts gemacht.