Contents

1 Einleitung

Im 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 einen *Semi-supervised* Ansatz umzusetzen und daneben KIs mit Hilfe der frameworks FlairNLP, AllenNLP und Huggingface zu programmieren.

2 FlairNLP

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.

2.1 Vorgehen

Damit einer Präposition ein Sinn zugeordnet werden kann, passieren einige Dinge. Der Hauptteil des Projekts besteht aus einer NLP-KI mit dem Flair-framework.
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.

2.2 Anwendung

Für eine reibungslose Anwendung des Programmes müssen einige Dinge beachtet werden, denn es kann immer nur eine Präposition gleichzeitig klassifiziert werden. Deshalb wird ein *Sequence Tagger* benötigt, der die Präposition markiert, die klassifiziert werden soll. Dieser ist im Abschnitt 2.3 erklärt.

2.2.1 Satz predicten

Um einen Satz zu predicten wird das script *predict.py* benötigt. Beim Aufrufen kann der Satz, der predictet werden soll mit übergeben werden. Wichtig dabei ist, dass der Satz zuvor mit dem *Sequence Tagger* (↑) präpariert wurde. Sollte dies nicht geschehen sein, kann ein fehlerhaftes Ergebnis ausgegeben werden.

2.3 Sequence Tagger

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.

2.3.1 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!

2.3.2 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.

2.3.3 Programm

DEr *Sequence Tagger* ist in dem script *flair_sequence_tagger.py* enthalten. Er ist so konstruiert, dass er in anderen scripts leicht importiert und verwendet werden kann. Weiterhin beinhaltet das script auch eine beispielhafte Anwendung.

- __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.

2.4 Projektdateien

Das Flair-Projekt umfasst hauptsächlich vier Dateien:

- Flair_prepare.py - Diese Datei dient der Vorbereitung der Daten.
- Dataset_class.py - Diese Datei enthält wichtige Klassen auf Basis des FlairNLP-Frameworks zum ertsellen des Korpus.

- flair_test_classification_model.py - Diese Datei dient zum erstellen des *text classifiers* und des Trainings dessen
- predict.py - Diese Datei enthält den *predictor*, bzw. dient zum testen des Endproduktes.

Zusätzlich zu diesen vier Dateien gibt es noch weitere Dateien, darunter der eine Datei zum *Sequence Tagger*, der in Abschnitt 2.3 erklärt ist.
Daneben existieren noch weitere Dateien, die ähnlich zu den vier genannten Dateien sind. Dateien, die ein "*predict*" enthalten, funktionieren wie *predict.py*, Dateien, die ein "*text_classification_model*" enthalten, funktionieren wie *flair_test_classification_model.py*, benutzen aber z.B. andere Embeddings.

2.4.1 Flair_prepare.py

Der erste Schritt befasst sich mit der Vorbereitung der Daten. Wir benutzen für unser Training die Daten des **SemEval 2007**, die Rund 16.000 Sätze beinhalten zu 34 verschiedenen Prepositionen. Diese Daten sind in xml-Format gegeben, weshalb sie in csv-Format geändert werden müssen.
Als erstes werden die xml-Dateien mit Hilfe der Python-library *xml.etree.ElementTree* ausgelesen. Dabei speichern wir die *senseid* und den Satz in einer Liste. Die *senseid* wird zeitgleich mit dem von Flair benötigten Zusatz *__label__* versehen. Fehlerhafte Einträge - solche, bei denen entweder der Sinn oder der Satz fehlt bzw. fehlerhaft ist - werden zudem aussortiert und deren *instanceid* zur Überprüfung ausgegeben.
Nachdem alle Einträge in die Liste eingefügt wurden, wird diese gemischt und anschließend in *Training*, *Dev* sowie *Test* Dateien **disjunkt** aufgeteilt (80% - 10% - 10%).

2.4.2 Dataset_class.py

Die Datei *Dataset_class.py* enthält die Klassen zum Erstellen des Korpus. Diese sind unverändert aus dem **GitHub Repository** von Flair übernommen. Für eine genaue Dokumentation dieser beiden Klassen verweisen wir auf die Dokumentation von FlairNLP. Sie sind auf csv-Dateien angepasst.

2.4.3 Flair_text_classification_model.py

Diese datei beinhaltet alle Einstellungen und Bauteile für den *Text classifier*.

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

# 1. get the corpus
corpus: Corpus = CSVClassificationCorpus('data/
    ', col_name_map)
print(Corpus)

# 2. create the label dictionary
label_dict = corpus.make_label_dictionary()
```

Als erstes wird der Korpus auf Basis der zuvor erstellten csv-Dateien und der angegebnen *col_name_map* erzeugt, sowie ein Dictionary des Korpus errichtet.

Flair bietet neben einem csv-Korpus auch andere Formate, wir haben uns aber für csv entschieden, da es ein gängiges Format ist und auch für andere Teilprojekte verwendet werden kann.

```
# instantiate one-hot encoded word embeddings
   with your corpus
hot_embedding = OneHotEmbeddings(corpus)

# init standard GloVe embedding
glove_embedding = WordEmbeddings('glove')

# document pool embeddings
document_embeddings = DocumentPoolEmbeddings([
   hot_embedding, glove_embedding],
   fine_tune_mode='none')
```

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[1].

```
# 5. create the text classifier
#classifier = TextClassifier(
   document_embeddings, label_dictionary=
   label_dict)
classifier = TextClassifier.load('resources/
   best-model.pt')
```

Nachdem die Embeddings erstellt wurden, kann der Classifier erzeugt werden. Alternativ kann natürlicha uch ein bestehender Classifier geladen werden. Wichtig ist nur, dass dieser Classifier auch die Daten lesen kann, also dem selben Grundaufbau folgt.

```
flair.device = torch.device('cuda:0')

# 6. initialize the text classifier trainer
trainer = ModelTrainer(classifier, corpus)

# 7. start the training
trainer.train('resources/hot_embed/',
              learning_rate=0.1,
              mini_batch_size=32,
              anneal_factor=0.5,
              patience=5,
              max_epochs=10)
```

Bevor das Training startet, wählen wir noch, dass auf einer GPU trainiert werden soll. Da Flair über Torch läuft, kann dies einfach über troch gewählt werden.
Anschließend erstellen wir den Trainer auf Basis des zuvor erzeugten oder geladenen Classifier und Korpus und beginnen das Training. Hierbei können

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

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.

2.4.4 predict.py

Der Predictor dient dazu, die Fertige KI anzuwenden. Das Programm kann dafür auch aus der Konsole mit Übergabeparameter verwendet werden. Der Übergabeparameter muss dabei ein String sein.
Alternativ kann das Programm auch ohne Übergabeparameter ausgeführt werden. In diesem Fall wird eine Reihe an Testdaten predictet.

3 AllenNLP

3.1 Vorgehen

3.2 Programm

4 Blabla