# Contents

**1 Einleitung** **2**

**2 FlairNLP** **2**
2.1 Vorgehen [Training] . . . 2
2.2 Anwendung . . . 2
2.2.1 Trainieren . . . 2
2.2.2 Satz predicten . . . 2
2.3 Sequence Tagger - Präpositionen markieren . . . 3
2.3.1 Anwendung . . . 3
2.3.2 Funktionsweise . . . 3
2.3.3 Programm . . . 3
2.4 Projektaufbau . . . 4
2.4.1 Aufbau des Classifiers und Korpus . . . 4
2.4.2 Aufbau des Trainings . . . 5
2.4.3 Aufbau des Predictors . . . 6

**3 AllenNLP** **7**
3.1 Vorgehen . . . 7
3.2 Programm . . . 7

**4 Blabla** **8**

# 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 [Training]

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.[1]

## 2.2 Anwendung

### 2.2.1 Trainieren

### 2.2.2 Satz predicten

Um einen Satz zu predicten, ist nicht viel erforderlich. Beim Aufrufen muss eine Liste von *Strings* übergeben werden und optional dazu ein tokenizer, der genutzt werden soll. Wurde beim erstellen des Objektes angegeben, dass nicht tokenisiert werden soll, so wird das tokenizer-argument ignoriert. Die

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

übergebenen Sätze dürfen dabei aber 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.

## 2.3 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.

### 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 *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.

## 2.4 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.

### 2.4.1 Aufbau des Classifiers und Korpus

**NOT YET UPDATED - OLD DESCRIPTION**

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[2].

```
# 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 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.

### 2.4.2 Aufbau des Trainings

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

### 2.4.3 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.

# 3 AllenNLP

## 3.1 Vorgehen

## 3.2 Programm

# 4 Blabla