Dokumentation
Abbreviations_Spelling_ASentenceLength_Unknown

Im Ordner ‚code.und.messungen' befindet sich ein Python Programm für die Anwendung von Metriken auf Korpora. Dieses enthält u. A. Methoden um den Anteil an Rechtschreibfehlern, unbekannten Wörtern und die durchschnittliche Satzlänge von Textkorpora zu ermitteln. Zusätzlich wurde eine Methode zur Erkennung von Abkürzungen mittels eines CRF-Klassifikators mithilfe von Stanford NER und StanfordCoreNLP hinzugefügt. Hinweis: Da für Windows x64 keine PyEnchant Version vorliegt, wurde dieses auskommentiert, kann aber je nach Umgebung wieder auskommentiert werden.

Im Ordner ‚code.und.messungen' befindet sich zusätzlich eine Version des Programms, welches CSV Dateien einlesen und analysieren kann. Hierzu muss die Eingabedatei auf ‚.csv' enden und es können weitere Parameter übergeben werden (siehe hierzu den Programmcode).

Die Korpora müssen zuerst heruntergeladen werden und im Ordner corpora abgelegt werden: siehe https://www.nltk.org/data.html
und [Gi11] Gimpel, Kevin; Schneider, Nathan; O'Connor, Brendan; Das, Dipanjan; Mills, Daniel; Eisenstein, Jacob; Heilman, Michael; Yogatama, Dani; Flanigan, Jeffrey; Smith, Noah A.: Part-of-speech Tagging for Twitter: Annotation, Features, and Experiments. In: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: Short Papers - Volume 2. HLT '11, Association for Computational Linguistics, Stroudsburg, PA, USA, pp. 42–47, 2011.

Für die Ausführung des Programms:
python xMain.py mc [Pfad zum Korpus] english (oder german)
„code.und.messungen" erzeugt zusätzliche Dateien im TEMP Ordner. Dieser befindet sich im Root von „code.und.messungen". Je analysierter Korpus entstehen dort 2 Dateien. Die Namensgebung der ausgegebenen Dateien ist nicht abhängig vom originalen Korpusname. Die erste Datei hat die Endung „.conllu" und ist die Datei, die nach dem durchlaufen der StanfordCoreNLP Pipeline entsteht. Diese beinhaltet im CONLL Format:
- Das Token
- Den POS-Tag (Universal POS Tag)
- Universal dependency parsed annotation
- Fake gold NER tags. Für die weitere Verarbeitung mit StanfordNER wird diese Spalte benötigt, da StanfordNER vorgegaukelt wird, dass die Eingabe ein Gold-Korpus mit Gold-Tags enthält. Durch die Eingabe dieser Testdatei, werden dessen Tokenisierung, POS-Tags usw. übernommen.

Diese Datei wird StanfordNER übergeben und daraus entsteht die zweite Datei die beinhaltet:
- Das Token
- Die Fake Gold-Tags
- Und die Abkürzungsannotationen (ABBR oder O) die von StanfordNER hinzugefügt worden sind

Weiterhin befinden sich im Ordner ‚scripts' hilfreiche Python scrips, die für die Erstellung und Umwandlung der Korpora verwendet wurden.
Nach der Erkennung von Abkürzungen wurden folgende Scripts für die weitere Auswertung verwendet. Dabei wird als Korpus die Ausgabe aus „code.und.messungen" verstanden, welcher Abkürzungsannotationen beinhaltet.
- „analyze_corpus_with_abbreviation_tags.py" – Überblick über die Häufigkeit erkannter Abkürzungen

Im Ordner ‚Stanford_Sourcecode\StanfordNER' befindet sich der Quellcode für **StanfordNER** das für die Erkennung von Abkürzungen abgeändert wurde indem Features und Annotationen hinzugefügt wurden sind.
Zu den abgeänderten Java Klassen zählen:
- Stanford_NER/src/edu/stanford/nlp/ie/NERFeatureFactory.java
- Stanford_NER/src/edu/stanford/nlp/ling/AnnotationLookup.java
- Stanford_NER/src/edu/stanford/nlp/ling/CoreAnnotations.java
- Stanford_NER/src/edu/stanford/nlp/sequences/ColumnDocumentReaderAndWriter.java
- Stanford_NER/src/edu/stanford/nlp/sequences/SeqClassifierFlags.java
- Stanford_NER/src/edu.stanford.nlp.sequences.CustomColumnDocumentReaderAndWriter

Für eine Erklärung für das Hinzufügen eigener Features siehe:
https://mailman.stanford.edu/pipermail/java-nlp-user/2011-December/001567.html oder Anhang
Die Datei ‚property_file_cloud.prop' beinhaltet die Konfigurationsparameter für die Erstellung des für das Erkennung von Auflösungen verwendetem CRF-Klassifikators.
Für das Trainieren eines neuen CRF-Klassifikators:
java –jar stanford_ner.jar –props property_file_cloud.prop
Die für das Trainieren des Klassifikators verwendeten Features finden sich in der properties file.
- useClassFeature=true – „Puts a prior on the classes which is equivalent to how often the feature appeared in the training data."
[https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/ie/NERFeatureFactory.html]
- useWord=false – nicht das eigentlich Token als Feature verwenden
- useTags=true – verwende POS-Tag des Worts
- useLength=true – verwende die Wortlänge
- useSpecialCharacters=true – sind in einem Token Sonderzeichen enthalten
- usePeriod=true – sind in einem Token eine oder mehrere Punkte enthalten
- usePhraseStructure=true – verwende dependency parsed annotation
- usePrevWithoutWord=true – verwende auch Features des vorherigen Tokens
- useNextWithoutWord=true – verwende auch Features des nachfolgende Tokens
- wordShape=chris1 – verwende den chris1 word shape classifier. Teilt Tokens in Klassen ein. Beispielsweise SYMBOL oder ALLCAPS
- printFeatures=true – schreibt die Features in eine Datei (printFeatures-true.txt)
- useVowelsConsonants=true – verwendet die die Abfolge von Vokalen, Konsonanten, Sonderzeichen und Nummern als Feature für die Wortform

Im Ordner „Stanford_Sourcecode\StanfordCoreNLP" befindet sich der Quellcode für StanfordCoreNLP. Innerhalb von **StanfordCoreNLP** wurde der WhitespaceLexer verändert. Wörter werden nicht mehr durch normale Leerzeichen getrennt, da beispielsweise NPS_Chat Korpus Tokens mit Leerzeichen enthält, sondern mittels Tabulator-Zeichen (‚\t'). Dadurch können bereits tokenisierte Texte StanfordCoreNLP übergeben werden um diesen Texten POS-Tags und dependency parsed Annotation hinzuzufügen. Siehe die veränderte Jflex Datei

„edu.stanford.nlp.process.WhitespaceLexer.flex“ aus der
„edu.stanford.nlp.process.WhitespaceLexer.java“ generiert wird.
Außerdem wurden auch die meisten Änderungen wie in Stanford_NER in den obigen aufgeführten Klassen durchgeführt um die ebenso die zusätzlichen Features und Annotationen zu unterstützen. Weiterhin wurden Models fürs POS-Tagging und dependency parsing für Englisch und Deutsch in die ausführbare JAR hinzugefügt. Dies sind Models die von der Stanford Webseite heruntergeladen werden können und diese befinden sich in:

- „[src/bin]\edu\stanford\nlp\models\pos-tagger“ für POS-Tagging
- „[src/bin]\edu\stanford\nlp\models\parser\nndep“ für depdency parsing

Für StanfordCoreNLP sollte bei der derzeitigen Cloud-Instanz mit 50Gb RAM ein Korpus maximal ungefähr 250.000 Tokens enthalten, da ansonsten der Speicher ggf überläuft.
Für diesen Prototyp wurde StanfordCoreNLP und StanfordNER bearbeitet (siehe oben).

Erstellung der Korpora für das Trainieren und Testen. Aus den verschiedenen Ressourcen (Brown, Industriedaten, usw.) wurden jeweils manuell von Anfang der jeweiligen Datensätze an nach Sätzen gesucht, die Abkürzungen enthalten und diese entnommen. Weiterhin wurde der PunktSentenceTokenizer von NLTK auf jede einzelne Ressource trainiert und für einige Abkürzung die von dem Tokenizer gefunden wurden, Beispielsätze aus den Ressourcen entnommen. Außerdem wurden zufällige Sätze aus den Ressourcen entnommen. Daraufhin wurde iterativ einige male ein NER Model trainiert und auf die Ressourcen angewendet. Für Wörter die dabei besonders häufig fehlerhaft als Abkürzung erkannt wurden, wurden dann Beispielsätze aus den Ressourcen entnommen um den entgegenzuwirken. Für jede einzelne Ressource entstanden so mehrere Korpora (je nachdem wie viele Iterationen durchgeführt worden sind) aus Beispielsätzen von denen jeder Token entweder als Abkürzung (ABBR) oder als keine Abkürzung (O) annotiert worden ist. Die einzelnen Korpora wurden dann weiterhin aufgeteilt in Trainings- und Testkorpora. Wobei die Aufteilung 2:8 beträgt. Also abwechselnd wurden 8 Sätze dem Trainingskorpus zugeteilt und 2 dann dem Testkorpus. Als letztes wurden dann alle Korpora für alle Ressourcen, aufgeteilt in Trainings- und Testkorpus zu einem gesamten entweder Trainings- bzw. Testkorpus zusammengefügt. Mit diesen Korpora wurde dann das abschließende NER Model trainiert und getestet.
werden und in eine MySQL Datenbank geschrieben werden.

Anhang Features Implementierung Stanford NER:
Quelle: https://mailman.stanford.edu/pipermail/java-nlp-user/2011-December/001567.html

„[…]> *Hi,*
>
> *After getting back into this follows my report about integrating new features to the CRF classifier.*
>
> *Once the necessary steps were clear, things weren't that hard. My training file was generated with an external application (RapidMiner data mining suite) and was already extended by the desired column.*
>
> *So I just had to follow the steps John outlined:*
>
> *- add new CoreAnnotation sub-class to edu.stanford.nlp.ling.CoreAnnotations (just copy an existing one and adjust name (and type if necessary)) - this class could also be added anywhere else, but collecting them in a single file makes definitely sense*
>
> *- create a link between the new annotation and the desired key for the map property in edu.stanford.nlp.ling.AnnotationLookup. This means just appending a single line to the KeyLookup enumeration*
>
> *- add a flag for the new feature to edu.stanford.nlp.sequences.SeqClassifierFlags. This is described inside the NERFeatureFactory javadoc in a very comprehensible way. In my case I just added a public Boolean field (public boolean useMarkupFeature = false;) and a new else clause to the setProperties method:*
> *else if (key.equalsIgnoreCase("useMarkupFeature")) {*
> *useMarkupFeature = Boolean.parseBoolean(val);*
> *}*
>
> *- the last and worst documented step was adding the feature extractor to edu.stanford.nlp.ie.NERFeatureFactory. The javadoc just says*
> *"Add code to NERFeatureFactory for this feature. First decide which classes (hidden states) are involved in the feature. If only the current class, you add the feature extractor to the featuresC code, if both the current and previous class, then featuresCpC, etc." - ok, this helps in choosing the right place where to add the extractor, but what should it look like? Having a look at the other methods I decided to simply add the feature value with an appendix to hopefully make the feature name unique (this is a requirement). So I just wrote:*
> *if (flags.useMarkupFeature) {*
> *featuresC.add(c.get(MarkupAnnotation.class) + "-MARKUP");*
> *}*
>
> *After writing all this I just noticed that you updated the FAQ already :-*
>
> *Perhaps you might add some additional information regarding the last step. Struggling with this I also asked myself how features actually look like and how they are used. I found the property printFeatures which creates two text files (one of them is empty) that just contains the different values for each feature used. When only enabling word and my custom markup feature (for random values "value1" - "value5") the printed features are:*
>
> *value3-MARKUP|C*
> *value4-MARKUP|C*
> *value3-WORD|C*
> *value1-WORD|C*
> *value0-WORD|C*

> *value4-WORD|C*
> *value2-WORD|C*
> *value0-MARKUP|C*
> *value1-MARKUP|C*
> *value2-MARKUP|C*
>
> *Is there some other way to get some information about the data being used for classification?*
> *I would also be glad if someone might explain how these features are used and maybe provide a link to the theoretical elements such as potential functions. Do they just combine actual feature values for each token?*
[…]“