Norges teknisk–naturvitenskapelige universitet
Institutt for datateknologi og informatikk

TDT 4310
Spring 2023

Lab 3

I was in touch with Tollef regarding issues that led to me being less than equipped, both in terms of time and knowledge to do this lab. I have tried my best to give answers, but I will expect them to be lacking. He told me to try to finish the lab, and that I instead of implementing the latter part could explain how I would do so. Because of this, no python file will be included in the delivery.

1 POS chunking

Chunking allows us to group words based on their POS tags. This is useful for finding meaningful groups of words, such as noun phrases (NP) or verb phrases (VP). Given the sentence The quick brown fox jumps of the lazy dog", we have the following POS tatgs (using the universal tagset):
('The', 'DET') ('quick', 'ADJ') ('brown', 'ADJ') ('fox', 'NOUN') ('jumps' 'VERB') ('over', 'ADP') ('the', 'DET') ('lazy', 'ADJ') ('dog', 'NOUN')
Answer the following (using any tools available from NLTK or spaCy):

1) *Create a chunker that detects noun-phrases (NPs) and lists the NPs in the sentence.*
- See 1.2) for chunker that detects NPs and VPs.

2) *Modify the chunker to handle verb-phases (VPs) as well.*

```
sentence = 'The quick brown fox jumps over the lazy dog.'
nlp = spacy.load('en_core_web_sm')
doc = nlp(sentence)
noun_phrases = [chunk for chunk in doc.noun_chunks]
verb_phrases = [possible_subject.head for possible_subject in
    ↪ doc
             if possible_subject.dep == nsubj and
                ↪ possible_subject.head.pos == VERB]
print(f'Noun phrases in the sentence: {noun_phrases}')
print(f'Verb phrases in the sentence: {verb_phrases}')
```

```
» Noun phrases in the sentence: [The quick brown fox, the lazy dog]
Verb phrases in the sentence: [jumps]
```

3) *Can chunking be beneficial in the context of a predictive keyboard? Why or why not?*
- It can for example be beneficial by suggesting common phrases, instead of just the phrases' beginnings. If the current text is "I slept with", a feisty predictive

keyboard could predict "your mom", instead of "your". This can also help keeping more relevant information for the prediction in memory, by reading phrases as single tokens.

2 Dependency parsing

Dependency parsing can be used to extract the grammatical structure of a sentence. A particularly useful feature is to fetch the subject/objects and the root of a sentence.
Use spaCy to inspect/visualize the dependency tree of the sentence The quick brown fox jumps over the lazy dog"and answer:

1) *What is the root of the sentence?*
- displaCy shows that 'jumps' is the root of the sentence, as all the syntactic dependencies derive from it.

2) *Write a function to get the subject and object of a sentence. Print the results for the sentence above.*

```
def q2():
    sentence = 'The␣quick␣brown␣fox␣jumps␣over␣the␣lazy␣dog.'
    nlp = spacy.load('en_core_web_sm')
    doc = nlp(sentence)
    # displacy.serve(doc, style='dep')
    subj = None
    obj = None
    for token in doc:
        if 'subj' in token.dep_:
            subtree = list(token.subtree)
            subj = doc[subtree[0].i:subtree[-1].i+1]
        elif 'dobj' in token.dep_:
            subtree = list(token.subtree)
            obj = doc[subtree[0].i:subtree[-1].i+1]
    print(f'subject:␣{subj},␣object:␣{obj}')
```

- There is no direct object in the sentence, so we get these results:
```
» subject: The quick brown fox, object: None
```

3) How could you use the relationships from dependency parsing in a predictive keyboard?
- You could disregard certain word classes that wouldn't make sense to be the next word. So if you keep track of all the relationships in the sentence in addition to the last couple of words / chunks, the predictor could disregard lines of words it had seen before with its current scope of words / chunks, because it knew that it syntactically would little sense.

3 External knowledge using Wordnet and Sentiwordnet

These are both lexical resources that can be used to extract more information from

a word than what is available in the text itself. Wordnet is a lexical database for the English language, which can be used to find synonyms, definitions and more. Sentiwordnet is a lexical database for words and their associated opinionated values. Early sentiment models relied on these rule-based approaches, but we have since moved on to machine learning. However, while simplistic, they can still prove to be valuable. A singular word can sometimes be considered strictly positive or negative for next-word prediction.
Solve the following tasks:

1) *Use Wordnet (from NLTK) and create a function to get all synonyms of a word.*
- I read the task as just listing the synonyms crowded together, even though they might not be synonyms between themselves. The word 'crowd' produces the output The word 'crowd' produces the output **{'crew', 'herd', 'bunch', 'crowd', 'crowd_together', 'push', 'gang'}**.

2) *Explore other features you can extract from Wordnet for the same word. Are there any features that could be useful for a predictive keyboard?*
- With the .pos() attribute getter you could get POS, if you need that. It could also use the synonyms from the previous task, to try and give you more varied writing.

3) Use Sentiwordnet (from NLTK) and create a function to get the sentence's sentiment: "Well, I don't hate it, but it's not the greatest!"

- Consider features from Wordnet to get a more reliable sentiment score.
- Can you think of a way to alter texts to handle negations? (such as not the greatest")
- Explain your approach

- I started with tagging the sentence. Then I lemmatized the words, and decided which ones were likely to give relevant information in the translate_tagfunction. I took the first hit in the SentiWordNet as the most likely to represent the lemma the best, and added its positive score and subtracted its negative score from a score counter. I divided the score on the lemmas used in the score, and determined if it was negative or positive. The function returns 'positive(SCORE)' or 'negative(SCORE)'.

Like mentioned in the beginning of this exercise sheet answer, I had little time to catch up on all the subject matter and to do the lab, and didn't get around to addressing the negation . I had some ideas, for instance that I could potentially chunk them together and switch the phrase with an antonym. Whis idea would probably have been okay to implement, however I think it would have counted too much/too little in the opposite direction. With negated statements it's usually that the more drastic the statement is, its negation is the less drastic in the opposite direction. 'Not super bad' = 'a bit good', not 'Not super bad' = 'Super good'. Same goes for weak statements, the weaker it is, its negation is probably all the more drastic in the opposite direction. 'It was not even a little bit good' = 'Really bad'.
In the end I thought I should deliver this and add my thoughts on the idea instead.

```
def get_sentiment(sentence):
```

```
def translate_tag(tag):
    if tag[0] == 'J': return wn.ADJ
    elif tag[0] == 'N': return wn.NOUN
    elif tag[0] == 'R': return wn.ADV
    else: return None

tokens = nltk.word_tokenize(sentence)
tagged = nltk.pos_tag(tokens)

score = 0
lemmatizer = WordNetLemmatizer()
applied_lemmas = 0
for word, tag in tagged:
    wn_tag = translate_tag(tag)
    if wn_tag not in [wn.NOUN, wn.ADJ, wn.ADV]:
        continue
    lemma = lemmatizer.lemmatize(word, pos=wn_tag)
    if not lemma:
        continue
    synsets = swn.senti_synsets(lemma, pos=wn_tag)
    try:
        ss = list(synsets)[0]
    except IndexError:
        continue
    score += ss.pos_score() - ss.neg_score()
    applied_lemmas += 1
return f'positive({score/applied_lemmas})' if score > 0
    ↪ else f'negative({score/applied_lemmas})'
```

4 Building a sentiment analysis model, and the implementation

Like mentioned in the beginning of this exercise sheet answer, I spoke with Tollef, and he said that if I didn't have time to do the implementation I should explain what I would have done instead of doing it.

For the sentiment analysis model, I would have went with the Twitter binary set. I would have lemmatized and stripped the text, and used the scores that came with the data to train the model to score words on what sentiment they have. I don't have the time to look closely in to the sklearn library, to see what premade models I could have used for it, how many layers and what types.

For the smart keyboard, I would have tried to implement the stuff I mentioned in part 1. That entails to suggest phrases instead of single words, by seeing if the ll-ookahead"words of the predictor makes sense as a phrase or not, depending on what is currently written. I would also have used the dependency parsing to remove unlikely suggested words from the list the predictor can choose from. To do this the current text would have to be tagged and I would need to code in what would syntactically make sense to come after the current text (or maybe rather what would not make sense). Lastly I would have used the sentiment analysis model on the current text to display a perceived sentiment to the user - so that the user knows whether or not they come across as positive or negative.