Matriculation Number : 190000096

Part-of-Speech Tagging Using Hidden Markov Models. Will the Use of Smoothing and UNK Tags Helps to Improve Accuracy?

06.March 2020

Abstract

This report reflects the work of the project named POS Tagging, Smoothing and Unknown Words. It first introduces the background knowledge of POS Tagging, Hidden Markov Model, Viterbi Algorithm and Unknown Words. Then it comes to some preparation steps, including splitting training set and testing set and evaluating performance. Project results are divided into two parts : the influence of smoothing and the influence of UNK Tags. Error analysis using confusion matrix is also included. Besides English, this project also investigates UNK Tags in other languages, namely Portuguese and German. This report ends with a conclusion.

INTRODUCTION

Part-of-Speech Tagging

Part-of-speech , also known as POS, allows people to extract relationships between a word and its neighbours (Lane, Howard & Hannes, 2019). It is indispensable in terms of parsing, information extraction and coreference resolution. POS is divided into two categories: closed class and open class. Closed classes are POS with relatively fixed membership (Jurafsky & Martin, 2018) , such as prepositions, conjunctions and conjunctions. Open classes are those with continually changed membership. Nouns, verbs, adjectives and adverbs are open classes. New words have always been created, such as "e-bike" (an electric bike) and "sproglet" (baby or small child). A word can have more than one possible POS. For example, the word "work" can be a Noun in " The workers quit work at noon" and also can be a Verb in " I work in London". POS tagging is the disambiguation process. It assigns a proper POS to a word (Jurafsky & Martin, 2018). It can be trained on a certain number of tagged sentences and tested on sentences within the same dictionary. In Python, spaCy has implemented POS tagging functions for English and other languages, such as German (Figure 1). SpaCy is faster and more accurate than many other libraries (Omran & Treude, 2017).

```python
import spacy
from texttable import Texttable

t = Texttable()
header = ["Word", "Tag", "Tag Explanation"]
t.header(header)

sp = spacy.load('de')
# English: In the future we will hardly go to shops.
sentence = sp(u"In der Zukunft werden wir kaum noch in Geschäfte gehen.")
for l in range (len(sentence)):
    row = [sentence[l], sentence[l].pos_, spacy.explain(sentence[l].tag_)]
    t.add_row(row)

print(t.draw())
```

```
Console
<terminated> d.py [/Library/Frameworks/Python.framework/Versions/3.8/bin
+------------+-------+-----------------------------------+
|   Word     |  Tag  |          Tag Explanation          |
+============+=======+===================================+
| In         | ADP   | preposition; circumposition left  |
+------------+-------+-----------------------------------+
| der        | DET   | definite or indefinite article    |
+------------+-------+-----------------------------------+
| Zukunft    | NOUN  | noun, singular or mass            |
+------------+-------+-----------------------------------+
| werden     | AUX   | finite verb, auxiliary            |
+------------+-------+-----------------------------------+
| wir        | PRON  | non-reflexive personal pronoun    |
+------------+-------+-----------------------------------+
| kaum       | ADV   | adverb                            |
+------------+-------+-----------------------------------+
| noch       | ADV   | adverb                            |
+------------+-------+-----------------------------------+
| in         | ADP   | preposition; circumposition left  |
+------------+-------+-----------------------------------+
| Geschäfte  | NOUN  | noun, singular or mass            |
+------------+-------+-----------------------------------+
| gehen      | VERB  | infinitive, full                  |
+------------+-------+-----------------------------------+
| .          | PUNCT | sentence-final punctuation mark   |
```

Figure 1 : spaCy tags a german sentence

Hidden Markov Model and Viterbi Algorithm for Part-of-Speech

The Hidden Markov Model (HMM) is "a probabilistic sequence model". It has been successfully used in speech recognition and handwritten script recognition (He, 1988). With a given sequence of units, HMM calculates the probability distribution of labels' sequence and chooses the best sequence (Jurafsky & Martin, 2018). In POS, HMM interprets inputted words as observed events and POS tags as hidden events. First-oder HMM indicates "the probability of a state depends only on the previous state and the probability of a word appearing depends only on its own tag" (Jurafsky & Martin, 2018). A solution to HMM is the Viterbi Algorithm, which is a dynamic programming algorithm. It, through maximising the posteriori probability, gives an optimal estimation of the tag sequence (He, 1988).

Dealing With Unknown Words

Unknown words are words occur infrequently, or do not occur, in the training corpus. This project regards words occur only once (hapaxes) and do not occur in the training corpus as unknown words. In the first 10000 sentences of the brown corpus, there are 12178 hapaxes (Figure 2). Nearly 60% are noun, 18% are verb, 15% are adjective and 3% are adverb. This means open class type accounts for most of the hapaxes (95.3%). Those unknown words affect the accuracy of POS tagging. One of the most common ways to deal with unknown words is to replace those words with UNK Tags. For example, if a word ends with "ing", then it is likely to be a verb or noun, so words with this pattern can be replaced by "UNK-ING".

```python
from nltk.corpus import brown
from nltk.probability import FreqDist

words = []
tags=[]

for sent in brown.tagged_sents(tagset="universal")[:10000]:
    words.extend([w for (w, t) in sent])
    tags.extend([t for (w, t) in sent])


fdist = FreqDist(words)
hapaxes = fdist.hapaxes()

print("There are",len(hapaxes),"hapaxes in first 10000 sentences of brown corpus")

noun=0
verb=0
adj=0
det=0
conj=0
pron=0
adv=0

for l in range (len(words)):
    if words[l] in hapaxes:
        if tags[l]=="NOUN":
            noun=noun+1
        if tags[l]=="VERB":
            verb=verb+1
        if tags[l]=="ADJ":
            adj=adj+1
        if tags[l]=="DET":
            det=det+1
        if tags[l]=="CONJ":
            conj=conj+1
        if tags[l]=="PRON":
            pron=pron+1
        if tags[l]=="ADV":
            adv=adv+1

print(noun,"noun\n",verb, "verb\n",adj,"adj\n",adv,"adv\n",det,"det\n",conj,"conj\n",pron,"pron")
```

```
Console
<terminated> d.py [/Library/Frameworks/Python.framework/Versions/3.8/bin/python3]
There are 12178 hapaxes in first 10000 sentences of brown corpus
7151 noun
 2183 verb
 1862 adj
 405 adv
 2 det
 2 conj
 7 pron
```

Figure 2: Hapaxes in brown corpus

EXPERIMENTAL SETUP

The data used for the project is taken from the Brown University Standard Corpus of Present-Day American English (Brown Corpus). It contains text from 500 sources including news, governments fiction and so on (Bird, Klein & Loper, 2009). There are overall 57340 tagged sentences. The first 10000 sentences are used as train corpus and the next 500 sentences are test corpus. The words and tags are stored in words_train, words_test, tags_train and tags_test separately.

To evaluate performance, the program contains a method called calculate_accuracy (Figure 3). The prediction (backpointer) is compared with the tag in the train set. Through counting the number of correct and false predictions, this method returns the correct percentage rate.

```python
# calculating accuracy
def calculate_accuracy (tag, backpointer):
    right = 0
    wrong = 0
    for l in range(len(backpointer)):
        if backpointer[l] == tag[l]:
            right = right + 1
        else:
            wrong = wrong + 1
    return " {:.2%}".format(right / (right + wrong))
```

Figure 3 :Performance Evaluation

RESULTS

Results for whether Witten-Bell smoothing improves the accuracy (smooth.py)

Witten-Bell smoothing, an instance of interpolation smoothing, is used to avoid zero-frequency events (Watanabe & Chien, 2015). The main idea is replacing zero-frequency events with the probability of events occurring only once (Yadav, Joglekar, et al. , 2010). However, does Witten-Bell smoothing actually improve the accuracy? Will people get the same result when only smoothing observation likelihood and only smoothing transition probability?

To oder to investigate the effect of Witten-Bell Smoothing, smooth.py contains 4 different Viterbi methods. Viterbi_method uses both smoothed transition probability and smoothed observation likelihood. It calculates Viterbi value as :

*num_pre = viterbi_pre[tag_pre] * **smoothed_transition_probability**(tag_pre).prob(tag) * **smoothed_observation_likelihood(**tag).prob(sentence[l])*

viterbi_method_with_smooth_transition method only uses smoothed transition probability. It calculates Viterbi value as :

*num_pre = viterbi_pre[tag_pre] * **smoothed_transition_probability**(tag_pre).prob(tag) * **cpd_tagwords**[tag].prob(sentence[l])*

The other two methods, namely viterbi_method_with_smooth_observation and viterbi_method_without_smooth, changes its Viterbi elements in the same way. This part of the program focuses on investigating the smoothing effect. As a result, no UNK tags are used in smooth.py.

The result of smooth.py is shown in Figure 4. As can be seen from the figure, with the use of Witten-Bell Smoothing, the accuracy reaches 93.94%. The accuracy is down to 45.42% without any smoothing method. It is interesting to note that whether the transition probability is smoothed does not affect accuracy at all. The improvement comes all from smoothed observation likelihood.

```
Console
<terminated> ds.py [/Library/Frameworks/Python.framework/Versions/3.8/bin/python3]
+-------------------------------------------------------------+----------+
|                        Smooth or Not                        | Accuracy |
+=============================================================+==========+
| smoothed transition probability and observation likelihood |  93.94%  |
+-------------------------------------------------------------+----------+
| smoothed transition probability                             |  45.42%  |
+-------------------------------------------------------------+----------+
| smoothed observation likelihood                             |  93.94%  |
+-------------------------------------------------------------+----------+
| without smooth                                              |  45.42%  |
+-------------------------------------------------------------+----------+
```

Figure 4: Running Result of smooth.py

Results for whether UNK Tags improves the accuracy (tagging.py)

The words, which occur no more than one time in the training corpus and have certain patterns, are replaced with UNK Tags. This project investigates three possible UNK Tags. The first one is UNK-CAP and it is applied via the method replace_with_UNKCAP. This method replaces words in hapaxes, which are capitalized but not at the beginning of a sentence, with "UNK-CAP" (Figure 5). Those words are likely to be proper nouns.

```
# replace words capitalized which occur only once and not at the beginning
def replace_with_UNKCAP (tags_words):
    tags_words = tags_words
    fdist = FreqDist(words_train)
    hapaxes = fdist.hapaxes()
    for l in range(len(tags_words)):
        if tags_words[l][1] in hapaxes or tags_words[l][1] not in words_train:
            if tags_words[l][1][0].isupper() and tags_words[l - 1][1] != "START":
                tags_words[l] = list(tags_words[l])
                tags_words[l][1] = "UNK-CAP"
                tags_words[l] = tuple(tags_words[l])
    return tags_words
```

Figure 5: Replace Words with "UNK-CAP"

The second method is called replace_with_UNKED, which replaces words ending with "ed" by "UNK-ED" (Figure 6). Similarly, the third one is replace_with_UNKING. This method substitutes "UNK-ING" for words ending with "ing" (Figure 7).

```
# replace words ending with "ed" which occur only once
def replace_with_UNKED (tags_words):
    tags_words = tags_words
    fdist = FreqDist(words_train)
    hapaxes = fdist.hapaxes()
    for l in range(len(tags_words)):
        if tags_words[l][1] in hapaxes or tags_words[l][1] not in words_train:
            if tags_words[l][1].endswith("ed"):
                tags_words[l] = list(tags_words[l])
                tags_words[l][1] = "UNK-ED"
                tags_words[l] = tuple(tags_words[l])
    return tags_words
```

Figure 6: Replace Words with "UNK-ED"

```python
# replace words ending with "ed" which occur only once
def replace_with_UNKED (tags_words):
    tags_words = tags_words
    fdist = FreqDist(words_train)
    hapaxes = fdist.hapaxes()
    for l in range(len(tags_words)):
        if tags_words[l][1] in hapaxes or tags_words[l][1] not in words_train:
            if tags_words[l][1].endswith("ed"):
                tags_words[l] = list(tags_words[l])
                tags_words[l][1] = "UNK-ED"
                tags_words[l] = tuple(tags_words[l])
    return tags_words
```

Figure 7: Replace Words with "UNK-ING"

Among 252295 words in first 15000 sentences of the brown corpus, 3548 words can be replaced by "UNK-CAP", 1043 words replaced by "UNK-ED" and 825 words can be replaced by "UNK-ING" (Figure 8). The number of "UNK-CAP" is significantly higher than the other two. The reason is, as discussed previously, nearly 60% hapaxes are noun and a word capitalized but not at the beginning of a sentence is very likely to be a noun.

```python
print("There are overall",len(words_train)+len(words_test),"words in the first 15000 sentences of brown corpus")
print("There are ",len(hapaxes), " hapaxes in the training set ")

tags_words_train1 = replace_with_UNKCAP(tags_words_train)
tags_words_test1 = replace_with_UNKCAP(tags_words_test)
print("There are",cap, "words being replaced by UNK-CAP")
tags_words_train2 = replace_with_UNKED(tags_words_train)
tags_words_test2 = replace_with_UNKED(tags_words_test)
print("There are",ed, "words being replaced by UNK-ED")
tags_words_train3 = replace_with_UNKING(tags_words_train)
tags_words_test3 = replace_with_UNKING(tags_words_test)
print("There are",ing, "words being replaced by UNK-ING")
```

Console

```
<terminated> ds.py [/Library/Frameworks/Python.framework/Versions/3.8/bin/python3]
There are overall 252295 words in the first 15000 sentences of brown corpus
There are  12178  hapaxes in the training set
There are 3548 words being replaced by UNK-CAP
There are 1043 words being replaced by UNK-ED
There are 825 words being replaced by UNK-ING
```

Figure 8: Words can be Replaced by UNK Tags

The program first tests separately whether these three UNK Tags can all improve accuracy. The result is shown in the Figure 9.

UNK-TAG	Accuracy
Without UNK TAG	93.94%
With UNK-ING	94.04%
With UNK-CAP	93.93%
With UNK-ED	94.16%

Figure 9 :First Part of tagging.py Output

The program successfully finds 93.94% of correct tags without using any UNK Tags. Accuracy improves to 94.16% by only using UNK-ED Tags and to 94.04% by only using UNK-ING Tags. Surprisingly, although the number of UNK-CAP Tags is significantly higher, introducing UNK-CAP Tags brings down the accuracy to 93.93%.

As can be noticed, there are some overlaps among these UNK-CAP and the other two UNK Tags. A capitalized word can also end with "ing" or "ed", such as "Republican-controlled" and "Campaigning". 90 words are involved (Figure 10).

```
n=0
def overlaps (tags_words):
    global n
    tags_words = tags_words
    fdist = FreqDist(words_train)
    hapaxes = fdist.hapaxes()
    for l in range(len(tags_words)):
        if tags_words[l][1] in hapaxes or tags_words[l][1] not in words_train:
            if tags_words[l][1][0].isupper() and tags_words[l - 1][1] != "START" and (tags_words[l][1].endswith("ed") or tags_words[l][1].endswith("ing")):
                n=n+1


tags_words_train = overlaps(tags_words_train)
tags_words_test = overlaps(tags_words_test)
print("There are",n, "words are involved")
```

```
Console
<terminated> ds.py [/Library/Frameworks/Python.framework/Versions/3.8/bin/python3]
There are 90 words are involved
```

Figure 10: Number of Overlaps

The second part of tagging.py explores whether the sequence of replacement with UNK Tags affects accuracy. 8 different combinations have been tested to find the best one. The output is shown in Figure 11. The best accuracy is highlighted with a red rectangle.

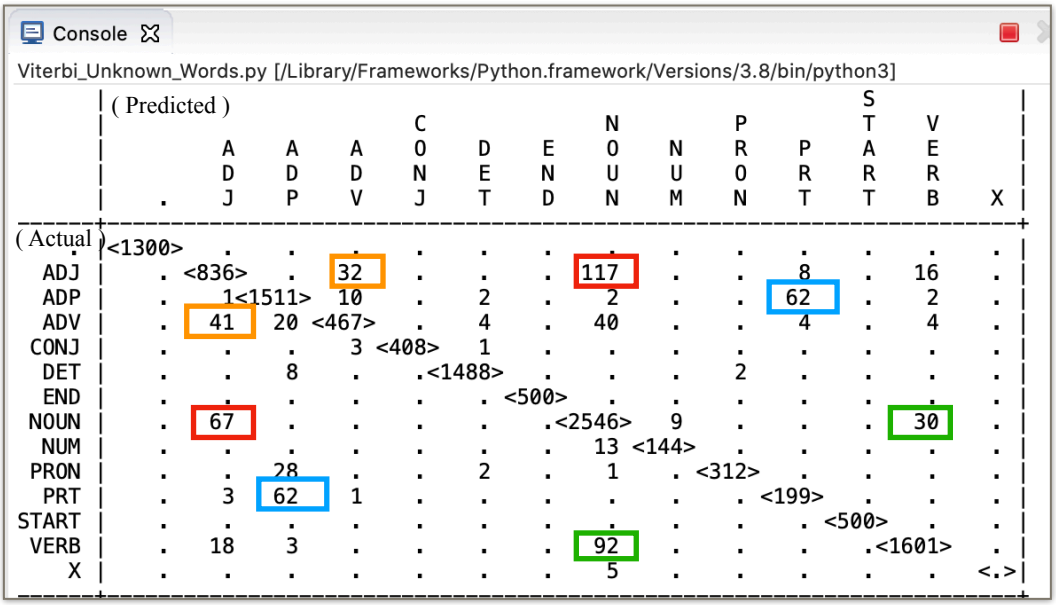
```
Console
<terminated> Viterbi_Unknown_Words.py [/Library/Frameworks/Py
+-----------------------------------+----------+
|              UNK-TAG              | Accuracy |
+===================================+==========+
| Without UNK TAG                   |  93.94%  |
+-----------------------------------+----------+
| With UNK-ING                      |  94.04%  |
+-----------------------------------+----------+
| With UNK-CAP                      |  93.93%  |
+-----------------------------------+----------+
| With UNK-ED                       |  94.16%  |
+-----------------------------------+----------+
| With UNK-ING and UNK-CAP          |  93.81%  |
+-----------------------------------+----------+
| With UNK-CAP and UNK-ING          |  94.03%  |
+-----------------------------------+----------+
| With UNK-ED and UNK-CAP           |  93.57%  |
+-----------------------------------+----------+
| With UNK-CAP and UNK-ED           |  94.06%  |
+-----------------------------------+----------+
| With UNK-ING and UNK-ED           |  94.31%  |
+-----------------------------------+----------+
| With UNK-CAP, UNK-ING and UNK-ED  |  94.22%  |
+-----------------------------------+----------+
| With UNK-ING, UNK-CAP and UNK-ED  |  94.24%  |
+-----------------------------------+----------+
| With UNK-ING, UNK-ED and UNK-CAP  |  94.24%  |
+-----------------------------------+----------+
```

Figure 11: The Output of tagging.py

The highest accuracy (94.31%) is reached by using only UNK-ING and UNK-ED. These two do not conflict with each other, so the sequence does not matter. Introducing UNK-CAP solely decrease accuracy, so it also brings down performance in combination with other UNK Tags. Using UNK-CAP, UNK-ING and UNK-ED simultaneously also improve accuracy to around 94.2%. The sequence of these three tags does not have a big impact on the final performance. The reason for this is that the number of overlaps is only 90 (as Figure 10), which accounts for such a small slice of the total words. As a result, the number of UNK Tags is not always directly proportional to accuracy. Some UNK Tags will lower the performance.

ERROR ANALYSIS

A Confusion matrix is a useful method to perform error analysis (Kulkarni & Shivananda, 2019). Tag A is confused with Tag B, if a word is predicted as Tag B where it should be Tag A. A confusion matrix reflects tags which are the most difficult to distinguish. With the use of Witten-Bell Smoothing and UNK Tags, the accuracy of POS tagging reaches 94.31%. 5.69% of the words are assigned with a false tag. The program tagging.py outputs a confusion matrix (Figure 12). Numbers embedded in "<>" are true positive, which means those words are tagged correctly. As can be seen from the confusion matrix, most of the confusion appears in pair. In other words, if words with Tag A is likely to be predicted as Tag B, then words with Tag B is also very likely to be predicted as Tag A.



Console

Viterbi_Unknown_Words.py [/Library/Frameworks/Python.framework/Versions/3.8/bin/python3]

(Actual) \ (Predicted)	.	ADJ	ADP	ADV	CONJ	DET	END	NOUN	NUM	PRON	PRT	START	VERB	X
.	<1300>	.	.	.	.	.	.	.	.	.	.	.	.	.
ADJ	.	<836>	.	32	.	.	.	117	.	.	8	.	16	.
ADP	.	1	<1511>	10	.	2	.	2	.	.	62	.	2	.
ADV	.	41	20	<467>	.	4	.	40	.	.	4	.	4	.
CONJ	.	.	.	3	<408>	1	.	.	.	.	.	.	.	.
DET	.	.	8	.	.	<1488>	.	.	.	2	.	.	.	.
END	.	.	.	.	.	.	<500>	.	.	.	.	.	.	.
NOUN	.	67	.	.	.	.	.	<2546>	9	.	.	.	30	.
NUM	.	.	.	.	.	.	.	13	<144>	.	.	.	.	.
PRON	.	.	28	.	.	2	.	1	.	<312>	.	.	.	.
PRT	.	3	62	1	.	.	.	.	.	.	<199>	.	.	.
START	.	.	.	.	.	.	.	.	.	.	.	<500>	.	.
VERB	.	18	3	.	.	.	.	92	.	.	.	.	<1601>	.
X	.	.	.	.	.	.	.	5	.	.	.	.	.	<.>

Figure 12 : Confusion Matrix of POS Tagging

There are four salient confusion pairs in the Figure 12. They are highlighted in 4 different colours. It is important to note that the confusion between noun and adjective occurs most frequently (in red rectangle). 117 words with the tag adjective have been wrongly predicted as noun and 67 noun have been tagged as adjective. Similarly, verb and noun (in green rectangle) are easily be confused. There are 92 verbs tagged as noun. There is also a chance of confusing adjective and adverb (in orange rectangle). Adjective, noun, verb and adverb are typical open class types. As a result, it can be derived that confusion happens among open class types more frequently. Open class types have larger lexicons and account for 95% of hapaxes. They are not as "stable" as closed class types. An interesting point: the blue rectangle indicates that the program tends to be confused between particle and adposition. Although these two are in the closed class team, they have some overlaps, such as "at" and "on". The tags of those words often strongly depend on the context. As a result, higher-order HMM could probably be used to further distinguish between particle and adposition.

UNK TAGS IN OTHER LANGUAGES

Compared to English, Portuguese and German are more complex. They have cases and genders. As discussed previously, using UNK Tags can improve accuracy in English. Does it also work in other languages?

UNK Tags in Portuguese (tagging_for_portuguese.py)

Floresta is a tagged Portuguese corpus in ntlk. This program uses the first 8700 sentences for training and the next 500 sentences for testing. The tags consist syntactic information, a plus sign and a conventional POS Tag (NLTK_Portuguese, 2020). A method called simplify_tag cut the syntactic part and the plus sign (Figure 13).

```
# simplify floresta tag
def simplify_tag(t):
    if "+" in t :
        return t[t.index("+")+1:]
    else:
        return t
```

Figure 13: Method simplify_tag

Two possible UNK tags are used to test the accuracy. One is UNK-CAP, which is same with UNK-CAP in tagging.py. Another one is called UNK-AR (Figure 14). In Portuguese, if a word is not capitalized and ends with "ar", it is likely a verb infinitive.

```
# replace words ending with "ar" which occur only once
def replace_with_UNKAR (tags_words):
    tags_words = tags_words
    fdist = FreqDist(words_train)
    hapaxes = fdist.hapaxes()
    for l in range(len(tags_words)):
        if tags_words[l][1] in hapaxes or tags_words[l][1] not in words_train:
            if tags_words[l][1].endswith("ar") and tags_words[l][1][0].islower():
                tags_words[l] = list(tags_words[l])
                tags_words[l][1] = "UNK-AR"
                tags_words[l] = tuple(tags_words[l])
    return tags_words
```

Figure 14: Replace with UNK-AR

As can be seen from Figure 15, accuracy increases from 92.34% to 94.67% with the use of UNK-CAP and to 92.55% when using UNK-AR. Because UNK-CAP and UNK-AR can improve the performance respectively, so the combination of UNK-CAP and UNK-AR is more effective than only using one. The result in Portuguese is same with that in English : If Tag A and Tag B can improve accuracy respectively, using Tag A and Tag B simultaneously will be more effective.

```
Console
<terminated> ddd.py [/Library/Frameworks/Python.framework/Versions/3
+----------------------+--------------------------+
|       UNK-TAG        | Accuracy for Portuguese  |
+======================+==========================+
| Without UNK TAG      |  92.34%                  |
+----------------------+--------------------------+
| With UNK-CAP         |  94.67%                  |
+----------------------+--------------------------+
| With UNK-AR          |  92.55%                  |
+----------------------+--------------------------+
| With UNK-AR, UNK-CAP |  94.88%                  |
+----------------------+--------------------------+
```

Figure 15 : Output of tagging_for_portuguese.py

Possible UNK Tags in German

German has three genders (masculine, feminine and neuter) and four cases (nominative, genitive, dative and accusative). In German, if a word is not capitalized and ends with "en", then it could be a verb infinitive or adjective, such as "geben (give)" and "guten Wein (accusative) (good wine)". Those can be replaced by "UNK-EN". Besides, UNK-CAP also works in German, but with a slight difference. If a word is capitalised and not at the beginning of a sentence, it is likely a noun, not necessarily a proper noun. This is because every noun is capitalised in German.

CONCLUSION

To sum up, without using any of the smoothing methods, the accuracy of POS tagging is less than 50%, because zero-frequency events largely affect the calculation of the Viterbi value. Witten-Bell Smoothing is an effective way to deal with zero-frequency events. The key to improving accuracy is introducing smoothed observation likelihood. Smoothed transition probability makes no difference to its accuracy.

UNK-ING and UNK-ED tags enhance the performance. With the use of these two UNK Tags, the accuracy increases to 94.31%, which is the highest accuracy among all combinations. However, UNK-CAP tags lower the accuracy to 93.93%. Because the number of words in this corpus, which are involved in overlaps between these three tags, is relatively small, so the sequence of UNK Tags is not an important factor influencing the performance in this project. As a result, it is not true that using more UNK Tags will definitely improve accuracy. Some UNK Tags can be counter-productive. The performance of each UNK Tag must be evaluated first.

Confusion happens more frequently among open class types, such as noun, verb, adjective and adverb. Besides, it is likely to wrongly tag particle and adposition because of the overlaps in their lexicon.

With the use of UNK-CAP and UNK-AR, the accuracy of tagging in Floresta rises from 92.34% to 94.88%. As a result, UNK-Tags also work in other languages. Introducing UNK Tags can improve accuracy, with the premise that those tags are tested respectively first.

Appendix A : Output of the Code

```
Console
<terminated> ds.py [/Library/Frameworks/Python.framework/Versions/3.8/bin/python3]
+-------------------------------------------------------------+-----------+
|                        Smooth or Not                        | Accuracy  |
+=============================================================+===========+
| smoothed transition probability and observation likelihood  |  93.94%   |
+-------------------------------------------------------------+-----------+
| smoothed transition probability                             |  45.42%   |
+-------------------------------------------------------------+-----------+
| smoothed observation likelihood                             |  93.94%   |
+-------------------------------------------------------------+-----------+
| without smooth                                              |  45.42%   |
+-------------------------------------------------------------+-----------+
```

Output of smooth.py

```
Console
<terminated> Viterbi_Unknown_Words.py [/Library/Frameworks/Python.framework/Versions/3.8/bin/python3]
+-----------------------------------+-----------+
|              UNK-TAG              | Accuracy  |
+===================================+===========+
| Without UNK TAG                   |  93.94%   |
+-----------------------------------+-----------+
| With UNK-ING                      |  94.04%   |
+-----------------------------------+-----------+
| With UNK-CAP                      |  93.93%   |
+-----------------------------------+-----------+
| With UNK-ED                       |  94.16%   |
+-----------------------------------+-----------+
| With UNK-ING and UNK-CAP          |  93.81%   |
+-----------------------------------+-----------+
| With UNK-CAP and UNK-ING          |  94.03%   |
+-----------------------------------+-----------+
| With UNK-ED and UNK-CAP           |  93.57%   |
+-----------------------------------+-----------+
| With UNK-CAP and UNK-ED           |  94.06%   |
+-----------------------------------+-----------+
| With UNK-ING and UNK-ED           |  94.31%   |
+-----------------------------------+-----------+
| With UNK-CAP, UNK-ING and UNK-ED  |  94.22%   |
+-----------------------------------+-----------+
| With UNK-ING, UNK-CAP and UNK-ED  |  94.24%   |
+-----------------------------------+-----------+
| With UNK-ING, UNK-ED and UNK-CAP  |  94.24%   |
+-----------------------------------+-----------+
The confusion matrix of POS tagging (Accuracy 94.31%) :
      |                                                                   S           |
      |                        C             N         P         T    V      |
      |         A    A    A    O    D    E    O    N    R    P    A    E      |
      |         D    D    D    N    E    N    U    U    O    R    R    R      |
      |    .    J    P    V    J    T    D    N    M    N    T    T    B    X |
------+-----------------------------------------------------------------------+
    . |<1300>   .    .    .    .    .    .    .    .    .    .    .    .    . |
  ADJ |    . <836>   .   32    .    .    .  117    .    .    8    .   16    . |
  ADP |    .    1<1511>  10    .    2    .    2    .    .   62    .    2    . |
  ADV |    .   41   20 <467>   .    4    .   40    .    .    4    .    4    . |
 CONJ |    .    .    .    3 <408>   1    .    .    .    .    .    .    .    . |
  DET |    .    .    8    .    .<1488>   .    .    .    2    .    .    .    . |
  END |    .    .    .    .    .    . <500>   .    .    .    .    .    .    . |
 NOUN |    .   67    .    .    .    .    .<2546>   9    .    .    .   30    . |
  NUM |    .    .    .    .    .    .    .   13 <144>   .    .    .    .    . |
 PRON |    .    .   28    .    .    2    .    1    . <312>   .    .    .    . |
  PRT |    .    3   62    1    .    .    .    .    .    . <199>   .    .    . |
START |    .    .    .    .    .    .    .    .    .    .    . <500>   .    . |
 VERB |    .   18    3    .    .    .    .   92    .    .    .    .<1601>   . |
    X |    .    .    .    .    .    .    .    5    .    .    .    .    .   <.>|
------+-----------------------------------------------------------------------+
(row = reference; col = test)
```

Output of tagging.py

```
Console
<terminated> ddd.py [/Library/Frameworks/Python.framework/Versions/3.8/bi
+----------------------+-------------------------+
|       UNK-TAG        | Accuracy for Portuguese |
+======================+=========================+
| Without UNK TAG      |  92.34%                 |
+----------------------+-------------------------+
| With UNK-CAP         |  94.67%                 |
+----------------------+-------------------------+
| With UNK-AR          |  92.55%                 |
+----------------------+-------------------------+
| With UNK-AR, UNK-CAP |  94.88%                 |
+----------------------+-------------------------+
```

Output of tagging_for_portuguese.py

Appendix B : REFERENCE

Bird, S., Klein, E. & Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly.

He, Y. (1988). *Extended Viterbi Algorithm for Second Order Hidden Markov Process*. [1988 Proceedings] 9th International Conference on Pattern Recognition, 718-720 vol.2.

Jurafsky, D. & Martin J.H. (2018). *Speech and Language Processing : An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Upper Saddle River, N.J.: Pearson Prentice Hall. 3rd Edition.

Kulkarni, A. & Shivananda, A. (2019). *Natural Language Processing Recipes*. APress.

Lane, H., Howard, C. & Hannes, M.H. (2019). *Natural Language Processing in Action*. Manning Publications Co. Shelter Island.

NLTK_Portuguese. http://www.nltk.org/howto/portuguese_en.html. [accessed on 09.03.2020]/

Omran, F.N., & Treude, C. (2017). *Choosing an NLP Library for Analyzing Software Documentation: A Systematic Literature Review and a Series of Experiments*. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 187-197.

Watanabe, S. & Chien, J. (2015). *Bayesian Speech and Language Processing*. Cambridge University Press.

Yadav, N., Joglekar, H. et al. (2010). *Statistical Analysis of the Indus Script Using n-Grams.* PloS ONE 5(3): e9506.