Part-Of-Speech Tagging Using Viterbi Model

D-C0-2764-9 Jiang Rui, D-C0-2785-7 Wang Xin Yu

University of Macau

CISC3025 Natural Language Processing

April 13, 2023

1. Introduction

This project aims to realize Part-Of-Speech (POS) tagging task with the Viterbi Model. The program can process sentences and assign tags to each word.

POS tagging is a crucial step in natural language processing. Its purpose is to determine the part of speech of each word in a sentence, such as nouns, verbs, adjectives, etc. This helps computers better understand the structure and meaning of sentences.

POS tagging also has many applications in natural language processing. For example, it can be used for grammar analysis to help computers better understand the structure of sentences. It can also be used in text classification, information retrieval, machine translation and other fields. In addition, POS tagging can also be used for speech recognition and text to speech systems, helping computers more accurately recognize and pronounce words.

2. Viterbi Algorithm

Viterbi algorithm is a dynamic programming algorithm, which is used to obtain the maximum posterior probability estimation of the most likely hidden state sequence (called Viterbi path) (Wikipedia contributors, 2023). It is also commonly used in speech recognition, speech synthesis, session segmentation, keyword detection, computational linguistics, bioinformatics and other fields.

To be more precise, if we have a set of states Q and an observation O, we attempt to find a state sequence that maximizes P (Q | O). Through conditional probability, we can convert P (Q | O) to P (Q, O)/P (O), but since P (O) does not involve changes in the state sequence, it is unnecessary to find P (O). We only need to find the state sequence that maximizes P(Q,O). (Butler, 2022b)

$$P(Q,O) = argmax\ P(O|Q)P(Q) = \prod argmax(o_i|q_i)P(q_i|q_{i-1})$$

2.1. Viterbi Algorithm for POS Tagging

The Viterbi algorithm can be used for part of speech tagging (POS tagging). In this case, we can treat the hidden state as a part of speech label and observe the result as a word. We can use the Viterbi algorithm to find the most likely part of speech label sequence for a given word sequence.

For example, suppose we have a sentence: "I like to eat apples." We can use the Viterbi algorithm to predict the most likely part of speech label for each word. In this case, the

Viterbi algorithm may predict that "I" is the pronoun, "like" is the verb, "eat" is the verb, and "apple" is the noun.

For each word in the tested sentences, we consider 3 values: the probability of the word being labeled as each tag, the probability of the word's tag being the suffix of the previous tag, and the probability of previous tags. We use the dynamic programming to achieve this:

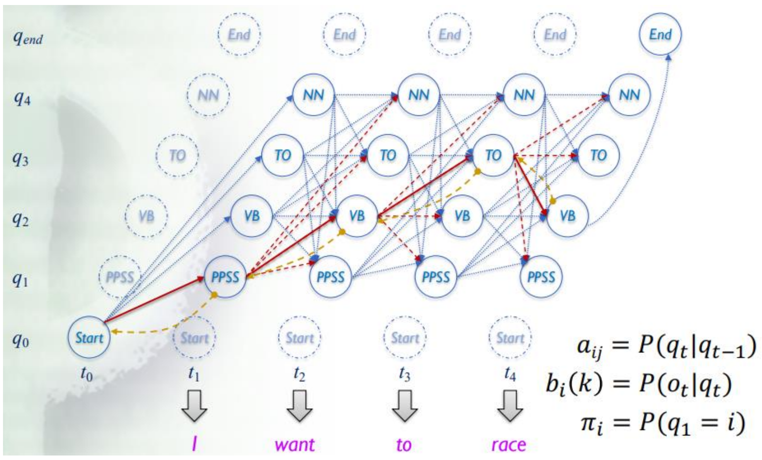


Figure 1 Viterbi encoding using dynamic programming.

We compute:

viterbi[*label*][*token*]=max P(*label* | *token*)P(*prev_label* | *label*)viterbi[*prev_label*][*prev_token*]

Each element in the viterbi matrix means “the maximum probability of the tag sequence from the start to the current word”. After finishing the computation, we use the backtrace to find the path. Firstly we start from the last time step, use the viterbi matrix to find the state with the highest probability, and record the corresponding tag as the tag of the last word of the sentence. Then, we find the best precursor state (max P(*label* | *token*)P(*prev_label* | *label*)) for the current state and record it as the tag of the word prior to the last word. This process continues until the first step.

3. Model design

3.1. Data

We use data from the Brown corpus(brown.sents()) and use the pos_tag() function from the nltk library to generate training data as well as the test data.

There are 57340 sentences with 1161192 tokens in total. The default ratio of test data is 10%, which means there are 5734 sentences of test data and 51606 sentences of train data.

We implement this approach to enlarge the size of the training set. The nltk.treebank() corpus provided by nltk library has POS tagging for training, but it is too small with merely less than 4,000 sentences, therefore to ensure our training set could be large enough for more features, we use the Brown corpus.

3.3. Feature Selection

Besides features needed in the Viterbi Algorithm, we also evaluate the other 24 features of tokens and vectorize tokens to feature vectors for the future procession. We show our feature pool in the table below.

Feature	meaning
is_punctuation	token is a punctuation(‘.’,’!’)
is_first_word	token is the first word of the sentences
is_last_word	token is the last word of the sentences
is_complete_capital	token consists of capital letters(ABC)
is_numeric	token consists of numbers(2008)
is_alphanumeric	token contains number(1st)
word_has_hyphen	token contains ‘-’
prefix_un-	token begins with ‘un’
prefix_in-	token begins with ‘in’
prefix_pre-	token begins with ‘pre’
prefix_dis-	token begins with ‘dis’

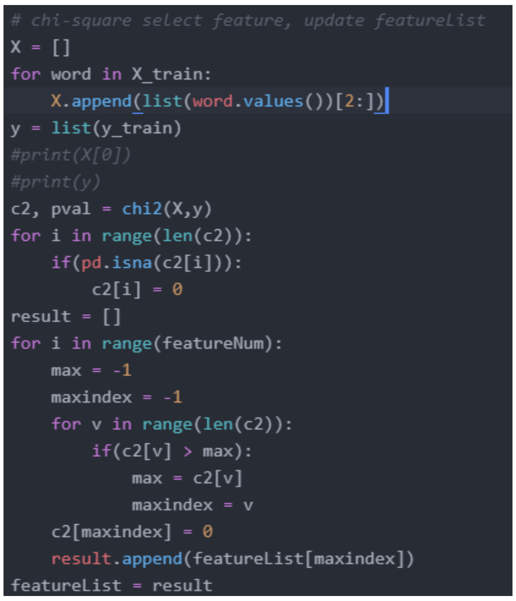
prefix_mis-	token begins with ‘mis’
prefix_non-	token begins with ‘non’
prefix_post-	token begins with ‘post’
suffix_-ed	token ends with ‘ed’
suffix_-ly	token ends with ‘ly’
suffix_-ing	token ends with ‘ing’
suffix_-ful	token ends with ‘ful’
suffix_-able	token ends with ‘able’
suffix_-less	token ends with ‘less’
suffix_-ness	token ends with ‘ness’
suffix_-men	token ends with ‘men’
suffix_-tion	token ends with ‘tion’

Then we use Chi-square test to rank all the features by its variation among labels, since not all the features are beneficial to the model.

Chi-square test is used to compare the goodness of fit between actual frequency and theoretical frequency. It is often used to analyze the relevance between two classification variables.(Cao & Zhou, 2020b)

In feature selection, the feature with higher chi-squared value means it is more dependent to the labels and therefore is a better feature for label classification.

In practice, we construct a matrix ‘X’ whose row vectors are feature vectors of each word in training set, and a vector ‘y’ whose elements are the corresponding tag of the words. Then we employ the chi2(X,y) function from the skitlearn library to compute the chi-squared values for each feature. Subsequently, we select the highest-ranking features based on their respective chi-squared values, in accordance with our specified selection criteria.



```
# chi-square select feature, update featureList
X = []
for word in X_train:
    X.append(list(word.values())[2:])
y = list(y_train)
#print(X[0])
#print(y)
c2, pval = chi2(X,y)
for i in range(len(c2)):
    if(pd.isna(c2[i])):
        c2[i] = 0
result = []
for i in range(featureNum):
    max = -1
    maxindex = -1
    for v in range(len(c2)):
        if(c2[v] > max):
            max = c2[v]
            maxindex = v
    c2[maxindex] = 0
    result.append(featureList[maxindex])
featureList = result
```

Figure 2 Calculate the chi-square value of each feature

3.3. Model Training

At the beginning of training, we first preprocess the training data in the following steps:

1. Untag training data: *untag()* separate the word and label from origin data.
2. Word to vector: *features()* turn each word to a feature vector.

Word vectorization (word vectorization) is a process of converting text data into numerical vectors so that machine learning algorithms can be used for processing. This transformation is usually achieved by mapping each word to a vector in high-dimensional space.

For example, the word "I" from the sentence "I love you." have features 'is_punctuation', 'is_first_word', 'is_last_word' and 'is_complete_capital' (meanings listed above). Then in the data, "I" would be a vector (0, 1, 0, 1). In this way, feature value is derived from word and the data storage structure also changed from data-sentence-word to data-vector, while the sentence information would be preserved in vector. And the whole sentence is a list of vectors shown below.

	is_punctuation	is_first_word	is_last_word	is_complete_capital
I	0	1	0	1

	is_punctuation	is_first_word	is_last_word	is_complete_capital
love	0	0	0	0
you	0	0	0	0
.	1	0	1	0

Then model training is processing the following data based on the training set.

1. The probability of each word being labeled as each label: *P(label | token)*
2. The probability of previous labels per pair: *P(prev_label | label)*
3. The probability of each feature appearing in each label: *P(feature | label)*
4. The probability of each label appearing in the training set: *P(label)*

Each of these probabilities is a model of a feature and is stored in a matrix. First two matrices are what we introduced in the basic Viterbi algorithm for POS Tagging, separately, they correspond to features *word* and *previous label*.

Then for the third kind of matrix, each additional feature would have a matrix for it. Matrix has structure *Mat[label][feature_value]*, to be more precise, it is the probability of each feature value appearing in each label, but as all additional features have boolean value 0 or 1, it is the same with the probability of each feature appearing in each label. Achieved by C(label, feature_value) / C(label).

For the fourth probability, it is used to replace *P(label | token)* for Out-Of-Vocabulary words which we will introduce later.

Emphasize that *P(label | token)* and *P(prev_label | label)* contains Laplacian Smoothing operation in order to deal with sparsity of training data. But for *P(feature | label)* it is not the case, as features here all have boolean value, we believe it's rather better to keep the value to 0 if there is not any this feature value occurs in this label in our training set. For example, in the case for the feature "prefix_un-" which is 1 only for words starting with "un" and label "." which is 1 only for full stop ".". Obviously, in our model there is P("prefix_un-" | ".") = 0, and when we are predicting a word with prefix "un", as we compute its Viterbi value for the label ".", the result will be 0 also, a detailed process for Viterbi value will be introduced in the next part. This reduces some computation and would benefit the accuracy.

3.4. Prediction

In testing, we construct a Viterbi matrix using dynamic programming as mentioned above. Additionally, we also consider additional features in our pool during the computation. Every selected feature would be considered here.

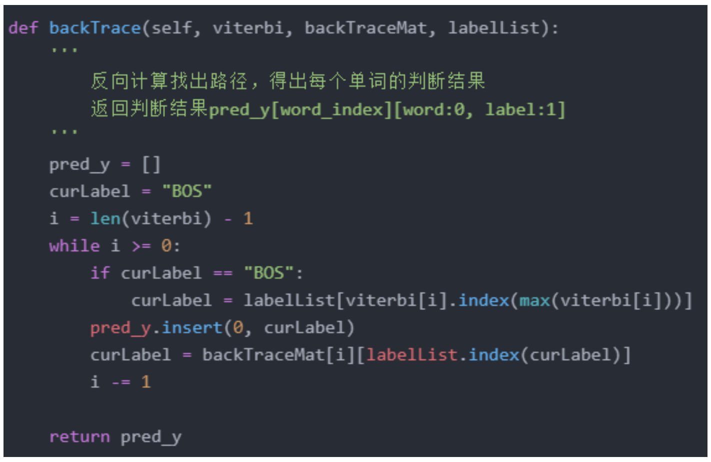
For each input sentence, like for training, we vectorize words of it, which convert each word into a feature sequence. Then for a feature sequence of a word, we implement the model to compute a Viterbi value for each label by the formula below.

$$\text{viterbi}[label][token]=\max\,(\text{P}(label \mid token) * \text{P}(prev\_label \mid label)) * \text{viterbi}[prev\_label][prev\_token] * \boldsymbol{\Pi}\ \mathbf{P}(\boldsymbol{feature} \mid \boldsymbol{label})$$

Emphasize that viterbi[*label*][*token*] is not similar with P(*label* | *token*), but the maximum probability of a sequence of labels, which end with the current label of the current word. Therefore a backtrace matrix backTraceMat[*token*][*label*] is also reserved. For each element within it means “the previous label which provides the maximum Viterbi value to the current label of the current word.”

Then we do the backtrace to get the label of each token with steps listed below.

1. Find the label with maximum probability from the Viterbi matrix in the last word of each sentences.
2. Start from the current label, store it into an output predict list, then use the backtrace matrix to find the previous label of the current one, then replace the current label with the found previous one iteratively.
3. Until the current label is the first word in a sentence, that is it has the previous label “BOS” in the traceback matrix.
4. Back to step 1.

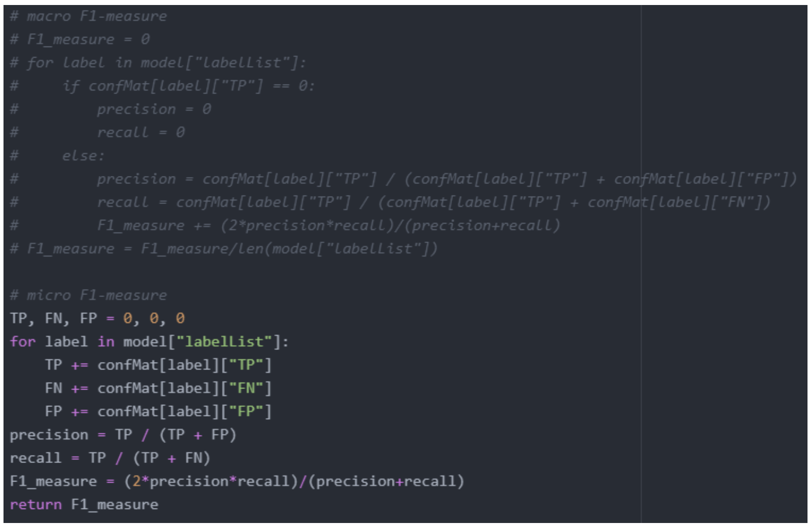


```python
def backTrace(self, viterbi, backTraceMat, labelList):
    '''
        反向计算找出路径，得出每个单词的判断结果
        返回判断结果pred_y[word_index][word:0, label:1]
    '''
    pred_y = []
    curLabel = "BOS"
    i = len(viterbi) - 1
    while i >= 0:
        if curLabel == "BOS":
            curLabel = labelList[viterbi[i].index(max(viterbi[i]))]
        pred_y.insert(0, curLabel)
        curLabel = backTraceMat[i][labelList.index(curLabel)]
        i -= 1

    return pred_y
```

Figure 3 Find path using backtrace.

After the prediction of each sentence, we compute the F1-score of the performance. Notice that here we only provide micro F1-score since macro F1-measure would be heavily affected by features with extremely high or low accuracy, thus couldn't reflect the performance of the model (macro F1-measure is also implemented but not used).



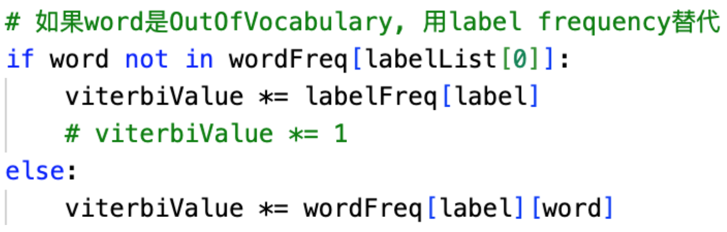
```python
# macro F1-measure
# F1_measure = 0
# for label in model["labelList"]:
#     if confMat[label]["TP"] == 0:
#         precision = 0
#         recall = 0
#     else:
#         precision = confMat[label]["TP"] / (confMat[label]["TP"] + confMat[label]["FP"])
#         recall = confMat[label]["TP"] / (confMat[label]["TP"] + confMat[label]["FN"])
#         F1_measure += (2*precision*recall)/(precision+recall)
# F1_measure = F1_measure/len(model["labelList"])

# micro F1-measure
TP, FN, FP = 0, 0, 0
for label in model["labelList"]:
    TP += confMat[label]["TP"]
    FN += confMat[label]["FN"]
    FP += confMat[label]["FP"]
precision = TP / (TP + FP)
recall = TP / (TP + FN)
F1_measure = (2*precision*recall)/(precision+recall)
return F1_measure
```

Figure 4 F1 score computation

3.5. Execution of Out-Of-Vocabulary Word

In the process of testing or predicting, there could be conditions that the appearing word is an Out-Of-Vocabulary (OOV) word. For our data set, in all 5,030,055 tested words there are 120,600 OOV words, which is 2.3976% of total words. In our Viterbi model, the appearance of OOV words mainly affects the feature matrix of feature *word*, which is we are not able to achieve P(*word*|*label*) through it. In this condition, we use the occurrence probability of each label P(*label*) to replace P(*oov*|*label*). It can be understanded as OOV words all have *word* feature as oov.



```python
# 如果word是OutOfVocabulary, 用label frequency替代
if word not in wordFreq[labelList[0]]:
    viterbiValue *= labelFreq[label]
    # viterbiValue *= 1
else:
    viterbiValue *= wordFreq[label][word]
```

Figure 5 Process the OOV words(tokens)

This process would result in a slightly better result in testing. Take our model viterboModel12 as an example, with this process, the F1-measure would increase from 0.9595 to 0.9597. It is a tiny improvement for two reason:

1. We have a large enough training set, so the OOV word only takes a small part of the tested word.
2. OOV words only affect a single feature, as we have several features in our model, the influence is reduced.

3.6. Testing Result

After comparing the F1-scores between different numbers of selected features(from 1 to 24), we find that when the number is around 13, the F1-score is the highest with 0.95972467173961 recorded.

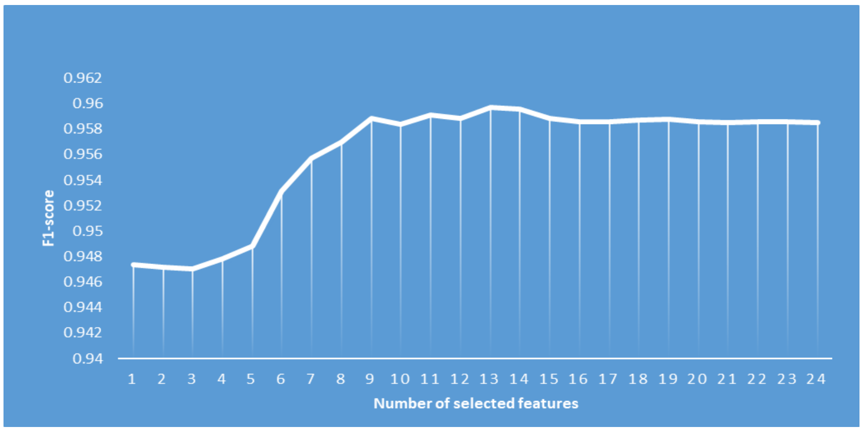


Figure 6 F1 scores among different numbers of selected features

From the graph we can see that, the

The labels sequence we sorted with Chi-Square is listed below,

order	label	Chi-Square
1	is_punctuation	909664.2012852447
2	is_last_word	870471.2315567094
3	is_complete_capital	811369.2317364215
4	suffix_-ing	656194.4864535753
5	is_first_capital	532165.3665627999

6	is_numeric	454719.8599158911
7	suffix_-ed	427964.583591518
8	suffix_-ly	252112.03752786893
9	word_has_hyphen	127054.12819388151
10	prefix_in-	92785.47621470148
11	is_first_word	82810.10807637646
12	suffix_-tion	57012.374173478915
13	suffix_-less	36680.18801229176
14	suffix_-able	23933.765304011755
15	suffix_-ment	17094.371037746307
16	suffix_-ful	9707.918258466054
17	prefix_un-	7808.070289696618
18	suffix_-ness	6797.766629944241
19	prefix_dis-	4357.749048578071
20	prefix_pre-	4310.805386184851
21	prefix_non-	1329.6364481218286
22	prefix_mis-	1011.1578577319957
23	is_alphanumeric	553.5742057395469
24	prefix_post-	397.9638002856705

From above data, we can observe that it is not the case that a feature evaluated good in Chi-square would perform better in the model. Instead, the first 5 features added into the model don’t improve its performance. But beginning from the sixth order feature, to the ninth order feature, when they are added into the model, its performance improves rapidly. However, we couldn’t conclude it is these features work better in our model, because the effect of features are cumulative.

And we try to analyze more detailed data. The below graph illustrates the change in precision, recall and F1-measure.

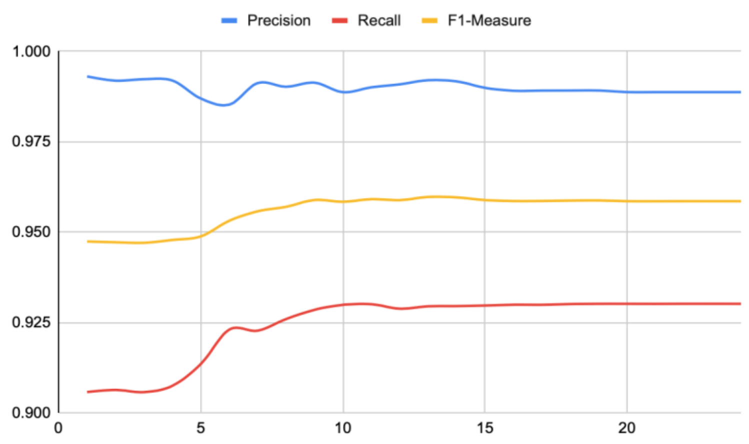


Figure 7 Precision, Recall and F1 score among different numbers of selected features.

It is clear that our model in testing has precision much higher than recall, and with more features being added, it mainly affects recall, and increment in recall results in the increment in F1-measure. Meanwhile, precision is relatively stable, which is floating around 0.99. And it has a high value in the beginning. We suggest it is probably because ordering features with Chi-Square contributes more on precision, that is why precision has a high initial value, and decreases when more features added and overall Chi-Square value decreases. For recall, what matters is the number of features, thus with more features added, recall value increases.

Then we show more detailed data, for labels in testing. Below is testing data of a few labels, we only select labels with high occurrence frequency here, as micro F1-measure is influenced more by frequent occurring labels.

POS	TP	FN	FP	Count	Precision	Recall	F1-measure
NN	14951	1206	46	16203	0.92535743	0.99693272	0.95981254
IN	13016	227	49	13292	0.98285887	0.99624952	0.98950889
DT	11465	118	47	11630	0.98981266	0.9959173	0.9928556
JJ	6621	1279	48	7948	0.83810127	0.99280252	0.90891619
NNP	6401	190	48	6639	0.97117281	0.99255699	0.98174847

The distribution of labels is not equal obviously, these top five labels occupy the domestic part of testing data set, and the model’s performance on them would generally decide the performance. We can see the model has good performance on label IN, DT and NNP, but for NN and JJ it’s not good enough, especially JJ. Here we put forth a guess to explain why JJ has a really low precision – only around 0.8381, it’s probably due to the features we selected. JJ is the POS label for adjective or numeral, ordinal, in our feature pool, there is hardly a feature related with these kinds of word, only suffix-less and suffix-able are features hold by JJ words, but only suffix-less is selected into the model. That may explain the reason why in the test set, label JJ has so many False Negative cases, which implies that the model always predicts a JJ word as other labels.

4. Execution

4.1. Environment

The programming environment is Python(3.8.3 or later), IDE we used includes Jupyter Notebook, Visual Studio Code,

4.2. Libraries

This project involves modules of nltk, sklearn, pandas, string, json, os, re and flask. Please make sure you have install the modules already, and follow the instructions if the system mentions that some packages or libraries need to be downloaded.

4.3. Running

The program files are stored in ‘projectProgram’ folder. The viterbiModel.py file contains the modularized code of the whole project. The run.py file contains setting up code for cmd commands. The app.py file contains codes for running local server for website

There are main two ways of running the program: running by cmd command and running through a webpage.

4.3.1. Running by cmd Command

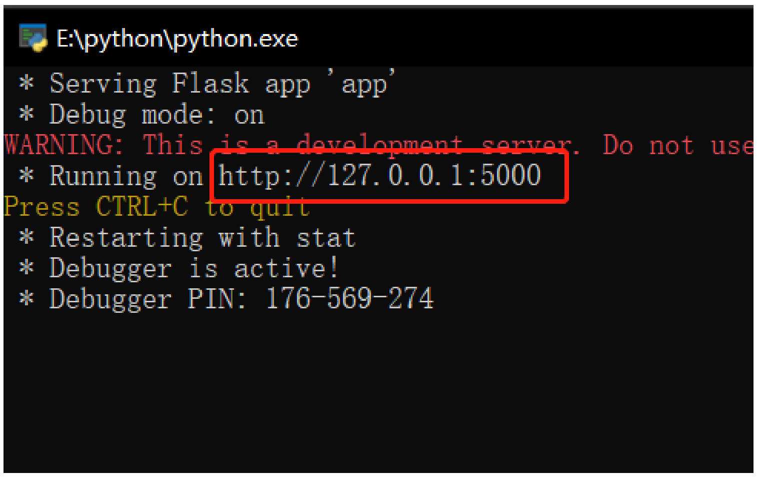
There are 4 cmd commands, belows are examples:

```
python run.py -t 13
//train vertibi model, input feature numbers 13
python run.py -d
//test current model, return f1 score
python run.py -dn viterbiModel13.txt
//test trained model, input model file name, return f1 score
python run.py -p “I like NLP!”
//predict part-of-speech tag of a single sentence, input a sentence
```

Make sure that you are under the same catalogue with the run.py file when you input the commands.

4.3.2. Running by local webpage

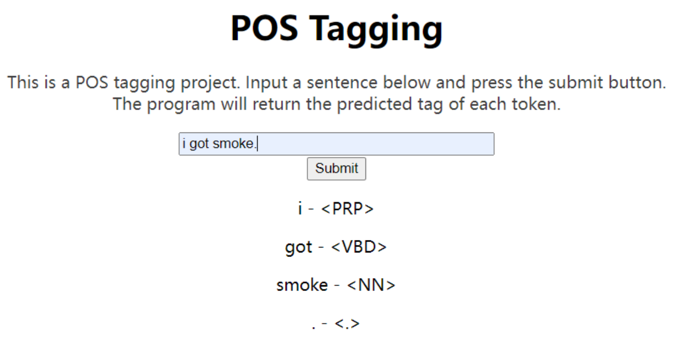
We made another interface for predicting a sentence through webpage. Firstly, you need to run the app.py file. A local server will be built and it return an IP address of the webpage



```
E:\python\python.exe
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 176-569-274
```

Figure 8 local server for the webpage.

Open this website in a browser and you can see a input component for inputting a sentence. After inputting a sentence and pressing the 'submit' button, the webpage will show the tag result of each tokens in the sentence.



POS Tagging

This is a POS tagging project. Input a sentence below and press the submit button.
The program will return the predicted tag of each token.

i got smoke.

Submit

i - <PRP>

got - <VBD>

smoke - <NN>

. - <.>

Figure 9 The project webpage.

5. Further Improvement

We have tried to perfect our project, but of course there is a lot to do that can improve our job.

1. Our feature pool can be further enlarged. We try to list all feature that may contribute to a NLP model, but still too little. A larger feature pool means it can possibly have more useful features.
2. Our model can be more complex also. In our model, we use Viterbi algorithm in bigram, which means we only consider the label of the previous word. For a more precise model, it could be a trigram or N-gram, that consider N previous word might bring more usable features and come to a better result. Also, it could consider the next word or next N words.
3. Our dataset is not for POS tagging, labels of the dataset are generated by nltk.pos_tag(), which could originally wrongly label some words. And as our model is trained through this dataset, some errors could be inherited.

6. Conclusion

In conclusion, it is a model that realizes Part-Of-Speech Tagging task with improvement in basic Viterbi Algorithm with more features are selected by Chi-Square and implemented in the model.

7. Reference

Wikipedia contributors. (2023, March 17). *Viterbi algorithm*. Wikipedia. https://en.wikipedia.org/wiki/Viterbi_algorithm

Butler, P. (2022b, March 30). *Intro to the Viterbi Algorithm - MLearning.ai - Medium*. Medium. https://medium.com/mlearning-ai/intro-to-the-viterbi-algorithm-8f41c3f43cf3

Cao, H., & Zhou, C. (2020b). Feature Selection Method Based on Chi-Square Test and Minimum Redundancy. *Advances in Intelligent Systems and Computing*. https://doi.org/10.1007/978-3-030-63784-2_22

Samet Çetin (2018, Jul 29). Part-Of-Speech (POS) Tagging - Medium
Medium
https://cetinsamet.medium.com/part-of-speech-pos-tagging-8af646a3d5bb