

## CSCE 5290 Natural Language Processing

### FINAL PROJECT

#### Twitter Parts of Speech tagging using Hidden Markov Models

##### Introduction:

In this project, I've used the twitter data that is taken from the following link: this link contains, the data that is present in the form of each word present in each line and I've to perform some preprocessing on it, to get the data in the format that is used to Homework 3. Where, all the data is present in the form of word/tag for each sentence/tweet and each sentence/tweet present in a separate line.

Source for data: <http://www.cs.cmu.edu/~ark/TweetNLP/>

The tag set for the twitter data contains, 25 distinct tags in it. 20 tags among them are all regarding the actual POS tags like Nouns, Verbs etc. and the remaining 5-6 tags are related to twitter data, like hashtags (#), usernames (@), Retweets (RT, etc.). The following table shows the available tags in this tag set.

TAG	Description	Contribution (%)
N	Nouns	12.82
O	Pronouns	6.72
S	Nominal + Possessives	0.11
^	Proper Nouns	5.78
Z	Proper Nouns + Possessives	0.14
L	Nominal + Verbal	1.53
M	Proper Nouns + Verbal	0.01
V	Verbs	14.30
A	Adjectives	4.79
R	Adverbs	4.32
!	Interjections	2.43
D	Determiners	5.69
P	Prepositions	8.15
&	Coordinating Conjunctions	1.59
T	Verb Particle	0.62
X	Pre-Determiners	0.09
Y	X + Verbal	0.01
#	Hashtags	0.93
@	Users	4.60
~	Retweets	3.41
U	URL	1.51
E	Emoticons	0.96
\$	Numerals	1.45
,	Punctuations	10.66
G	Others	0.97

**Data files:**

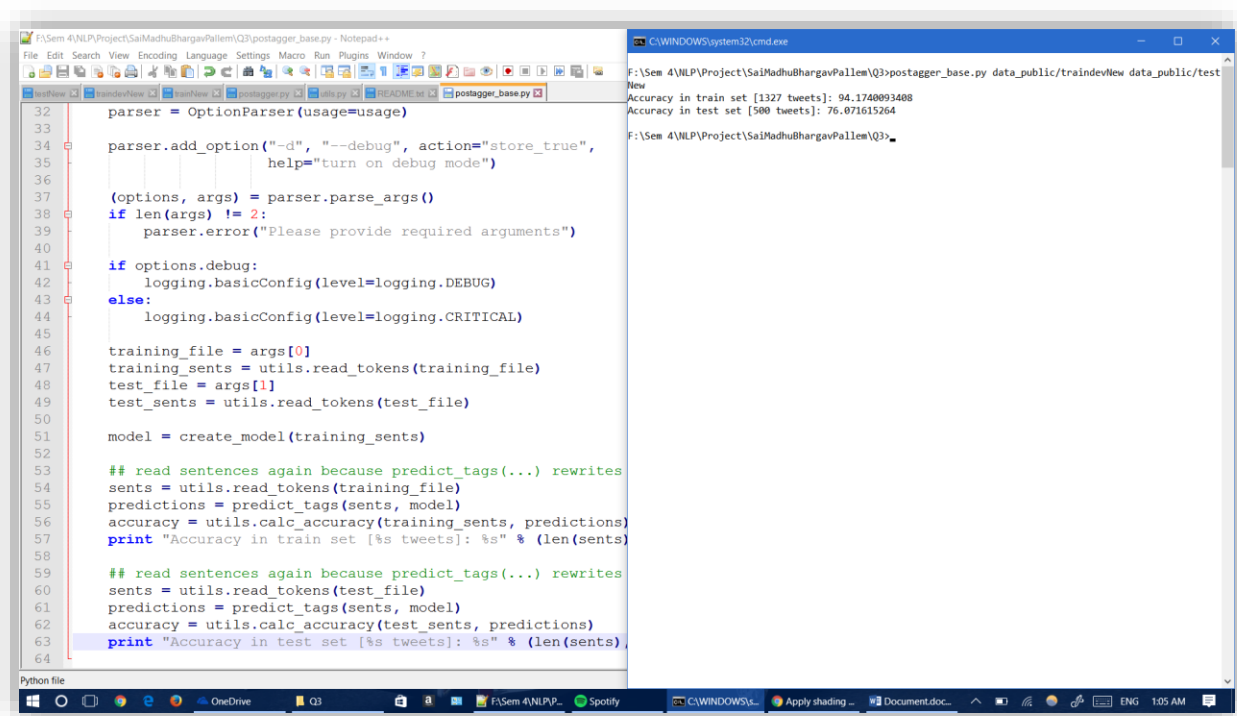
Two files are used to for this implementation, one used for training the model and another is used to test the model. The train set used contains, 1327 tweets and the test document consists of 500 tweets.

**Baseline Implementation:**

On performing the baseline tagger for the twitter dataset, the results are as follows for both training set and the test set:

Accuracy in train set [1327 tweets]: 94.1740093408

Accuracy in test set [500 tweets]: 76.071615264



```
32 parser = OptionParser(usage=usage)
33
34 parser.add_option("-d", "--debug", action="store_true",
35                  help="turn on debug mode")
36
37 (options, args) = parser.parse_args()
38 if len(args) != 2:
39     parser.error("Please provide required arguments")
40
41 if options.debug:
42     logging.basicConfig(level=logging.DEBUG)
43 else:
44     logging.basicConfig(level=logging.CRITICAL)
45
46 training_file = args[0]
47 training_sents = utils.read_tokens(training_file)
48 test_file = args[1]
49 test_sents = utils.read_tokens(test_file)
50
51 model = create_model(training_sents)
52
53 ## read sentences again because predict_tags(...) rewrites
54 sents = utils.read_tokens(training_file)
55 predictions = predict_tags(sents, model)
56 accuracy = utils.calc_accuracy(training_sents, predictions)
57 print "Accuracy in train set [%s tweets]: %s" % (len(sents),
58
59
60 ## read sentences again because predict_tags(...) rewrites
61 sents = utils.read_tokens(test_file)
62 predictions = predict_tags(sents, model)
63 accuracy = utils.calc_accuracy(test_sents, predictions)
64 print "Accuracy in test set [%s tweets]: %s" % (len(sents),
```

```
F:\Sem 4\NLP\Project\SaiMadhuBhargavPalleM\Q3>postagger_base.py data_public/traindevNew data_public/testNew
Accuracy in train set [1327 tweets]: 94.1740093408
Accuracy in test set [500 tweets]: 76.071615264
F:\Sem 4\NLP\Project\SaiMadhuBhargavPalleM\Q3>
```

We can improve the performance of this baseline by training the model using a bigram model. The following sections include the work that I've done regarding the same.

**Bigram implementation of Twitter POS tagging:**

To perform the bigram implementation on this data, I've made use of the postagger.py file that I've written for the homework 2. But, upon performing the bigram implementation, the results are as follows:

Accuracy in train set [1327 tweets]: 96.5381096827

Accuracy in test set [500 tweets]: 21.9289074752

The screenshot shows a Windows desktop with two windows. The left window is a Notepad++ editor displaying a Python script named 'postagger.py'. The script is an English Language POS Tagger that uses a Viterbi algorithm to tag words in a sentence. It includes a list of regular expressions for various parts of speech like hash, user, num, url, emoticon, proper, verb, adj, adverb, inter, abb, and retweets. The right window is a Command Prompt showing the output of the script. It displays the training and testing accuracies for the tagger.

```

F:\Sem 4\NLP\Project\SaiMadhuBhargavPallem\Q3>postagger.py data_public/traindevNew data_public/testNew

ENGLISH LANGUAGE POS TAGGER

Enter which smoothing to apply
1) Laplace (Add 1)
2) Backoff
3) Linear Interpolation

Enter your choice:
3
Training Completed
Should wait around 4 mins for every 10,000 sentences

Accuracy in train set [1327 tweets]: 96.5381096827
Accuracy in test set [500 tweets]: 21.9289074752

F:\Sem 4\NLP\Project\SaiMadhuBhargavPallem\Q3>

```

If we see this, the performance is very poor, i.e. the accuracy for test set is around 20 %. This is because, the twitter data set is the most ambiguous data set as there will be new hashtags and new user reference and as it is an open dataset, we must expect new words in every other tweet. To overcome this, we should come up with some morphological rules to get better accuracies. As the data set contains more Nouns, by default if we set all the unknown words to noun we get the following accuracies.

Accuracy in train set [1327 tweets]: 96.5381096827

Accuracy in test set [500 tweets]: 76.4113957135

To further improve these results, I've written the following morphological rules. The following section deals about various morphological rules that I've written and what is their contribution to the improvement in the accuracy.

### Morphological Rules:

The following are the set of rules are implemented in this code, to handle unseen words in the model. To handle them effectively, I came up with some regular expressions that will match various words in the testing phase that are not seen during the training phase and assign appropriate tags to them.

For this, all the regular expressions that I wrote will match each unseen word and if the regular expression matches it, then the probability of the word belonging to that tag is assigned with some value other than being a zero. This helps in further moving in the Viterbi instead of

just continuing filling the Viterbi matrix with zeros wherever there is an unknown word encountered. The following are the Morphological rules that I've implemented:

```
• hash = re.match('^#.*',word)
• user = re.match('^@.*',word)
• nums = re.match('.*[0-9].*',word)
• urls = re.match('(^\http.*)|(^www.*)',word)
• emoticon = re.match(':.:',word)
• proper = re.match('[A-Z]',word)
• verb = re.match('.*ing$|.*ed$|.*in$|.*\'t$|.*\'nt$',word)
• adj = re.match('.*ed$', word)
• adverb = re.match('.*ly$', word)
• inter = re.match('[A-Z]{2,5}$', word)
• abb = re.match('([a-zA-Z])(\1)+', word)
• retweets = re.match('\\.\\.\\.\\.\\.\\.', word)
```

These morphological rules are used to identify various tags from the unknown words based on the pattern the word is in.

**hash = re.match('^#.\*',word)**

This is used to match words with hashtags.

Eg: #BuzzNtheBurgh, #RTthis, etc.

**user = re.match('^@.\*',word)**

This is used to match usernames in twitter data.

Eg: @deefizzy, @WhiteHouse, etc.

**nums = re.match('.\*[0-9].\*',word)**

This is used to match Numbers.

Eg: 123559, 2<sup>nd</sup>, etc.

**urls = re.match('(^\http.\*)|(^www.\*)',word)**

Used to match URLs or websites.

Eg: <http://cli.gs/E4rms>, etc.

**emoticon = re.match(':.:',word)**

Used to match emoticons that are unseen

Eg: :x, :P, etc.

**proper = re.match('[A-Z]',word)**

Used to match Proper Nouns.

Eg: Obama, UN, UK, etc.

**verb = re.match('.\*ing\$|.\*ed\$|.\*in\$|.\*\'t\$|.\*\'nt\$',word)**

Used to match verbs in past or past participle form.

Eg: running, wouldn't, etc.

**adj = re.match('.\*ed\$', word)**

Used to match adjectives that end in ed.

Eg: scared, etc.

**adverb = re.match('.\*ly\$', word)**

Used to match adjectives that end up in ly.

Eg: Parallely, willingly, etc.

**inter = re.match('^[A-Z]{2,5}\$', word)**

This is to match interjections.

Eg: WTH, OMG, etc.

**abb = re.match('([a-zA-Z])(\1)+', word)**

This is also to match some interjections that have some set of characters repeating for few times.

Eg: ommmmmmmg, FTWWWWWWWW, etc.

**retweets = re.match('\.\.\.', word)**

This is to match some retweet tags that contain three dots(.).

Eg: ..., .., etc.

Also if the unknown word is present in first 5 positions in the tweet it is most likely to be a proper noun, I've observed this by going through the data on and on. So I've also came up with a rule such that if the word is present in the first 5 positions, then it is given a probability of being a Proper noun.

The results are modified for these Morphological rules as follows:

Morphological Rule	Improvement in Accuracy
hash = re.match('^#.*',word)	0.5750130684
user = re.match('^@.*',word)	14.3491897543
nums = re.match('.*[0-9].*',word)	1.4113957135
urls = re.match('(^http.*) (^www.*)',word)	0.0522739153
emoticon = re.match(':.+',word)	0.0130684788
proper = re.match('[A-Z]',word)	17.1327757449
verb = re.match('.*ing\$   .*ed\$   .*in\$   .*t\$   .*nt\$',word)	1.9733403032
adj = re.match('.*ed\$', word)	1.0846837428
adverb = re.match('.*ly\$', word)	0.4704652378
inter = re.match('^([A-Z]{2,5})\$', word)	0.0055216315
abb = re.match('([a-zA-Z])(\1)+', word)	0.0392054365
retweets = re.match('\\.\\.\\.\\.+', word)	0.0013543568

Finally, on applying all these Morphological rules, the following are the results for the twitter dataset:

Accuracy in train set [1327 tweets]: 96.5921433744

Accuracy in test set [500 tweets]: 86.0157016683

```

46 wordTagCount[sentence[i].word][sentence[i].tag] += 1.0
47 #if sentence[i].tag == ',':
48 #    print sentence[i].word
49 tagCount[sentence[i].tag] += 1.0
50 wordList[sentence[i].word] += 1.0
51
52 for sentence in sentences:
53     for i in range(len(sentence)):
54         if i < len(sentence) - 1:
55             tagCountBigram[sentence[i].tag][sentence[i+1].tag] += 1.0
56
57 for key1, value1 in wordTagCount.iteritems():
58     for key2, value2 in value1.iteritems():
59         if wordList[key1] == 1:
60             singleTimeWords += 1.0
61             wordTagProb[key1][key2] = wordTagCountBigram[key1][key2] / (wordTagCount[key1] + len(tagCount))
62
63 noTags = len(tagCount)
64
65 lambdas = estimateLambdas(tagCount, tagCountBigram)
66
67 global smoothing
68 if smoothing == '2':
69     #Laplace
70     for key1, value1 in tagCount.iteritems():
71         for key2, value2 in tagCount.iteritems():
72             tagProbBigram[key1][key2] = (tagCountBigram[key1][key2]+1)/(tagCount[key1] + len(tagCount))
73 elif smoothing == '1':
74     #Linear Interpolation
75     lam1 = lambdas[0]
76     lam2 = lambdas[1]
77     for key1, value1 in tagCount.iteritems():
78         for key2, value2 in tagCount.iteritems():

```

TWITTER DATA POS TAGGER  
 -----  
 Enter which smoothing to apply  
 1) Linear Interpolation  
 2) Laplace (Add 1)  
 Enter your choice:  
 1  
 Training Completed  
 Should wait around 4 mins for every 10,000 sentences  
 Accuracy in train set [1327 tweets]: 96.5921433744  
 Accuracy in test set [500 tweets]: 86.0157016683  
 F:\Sem 4\NLP\Project\SaiMadhuBhargavPallem\Q3>  
 F:\Sem 4\NLP\Project\SaiMadhuBhargavPallem\Q3>

I've implemented two smoothing techniques, Linear Interpolation and Laplace add 1 smoothing. The results for both these smoothing techniques are as follows:

Smoothing	Accuracy
Linear Interpolation	86.0157016683
Laplace - Add 1	85.9789008832

This above table shows that upon implementing Linear Interpolation instead of Laplace Add 1 smoothing, we can get a performance boost of about, 0.0368007851%.

**Conclusion:**

Finally, we can say that, twitter data set is one of the dataset where most of the data is ambiguous and we can expect an unseen word for every 5 words. To overcome this, we must come up with morphological rules to handle these unseen words. Upon performing a Bigram HMM tagger on this data set and applying a Linear Interpolation smoothing technique, and writing about 13 Morphological rules, we can say the performance had improved a lot compared to the baseline tagger.