Course Project Report

Marmara University, CSE 4094 Natural Language Processing

MARMARA
UNIVERSITY

Farid Yagubbayli **150113901**

Burak Aybar **150112001**

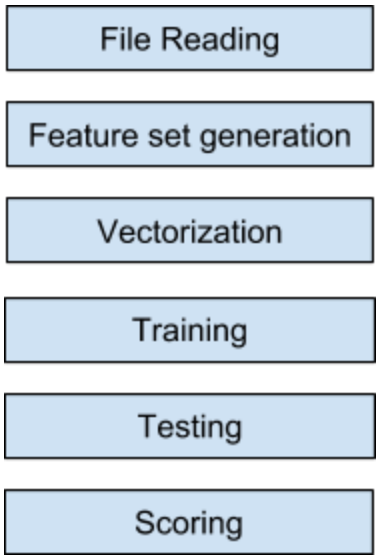
23.11.2016

INTRODUCTION

This document aims to give a brief information about the course project of CSE4094 Natural Language Processing by explaining steps needed to develop the resulting system

In the project we are expected to develop a system based on machine learning. The system is expected to classify the words by saying the position of the word in the phrase and the type of the phrase itself. This will be done by looking to the words and/or their POS tags as well as the POS tags of previous words.

The general flow schema of the resulting system is as follows:



Following sections are giving a brief explanation for different parts of the solution.

Data Loading and Storage

System will learn on training set and will be able to classify instances from test set. These datasets consist of sentences where each word has a *Part of Speech (POS)* and *Chunk* tags.

```
exports NNS B-NP
. . O

At IN B-PP
the DT B-NP
same JJ I-NP
time NN I-NP
, , O
he PRP B-NP
remains VBZ B-VP
fairly RB B-ADJP
pessimistic JJ I-ADJP
about IN B-PP
the DT B-NP
outlook NN I-NP
for IN B-PP
imports NNS B-NP
, , O
given VBN B-PP
continued VBD B-NP
high JJ I-NP
consumer NN I-NP
```

The data files are structured in three sections - word, POS tag, Chunk tag. We are particularly interested on Chunk tags. Every Chunk tag consists of two parts. First part indicates whether the word is a beginning of a phrase ("B-") or it is in the middle of the phrase ("I-"). And the second part indicates the type of the phrase.

Also as you can observe there are empty lines and "O" type labels. Empty lines have a role as a separator between sentences where "O" labels indicate that the symbol is not a phrase. This are very helpful features of datasets and will be used during pre-processing stage.

All of above was implemented at *fileToSet* function which takes path of the data file and produces a list of sentences where each word has POS and Chunk tags.

Pre-Processing

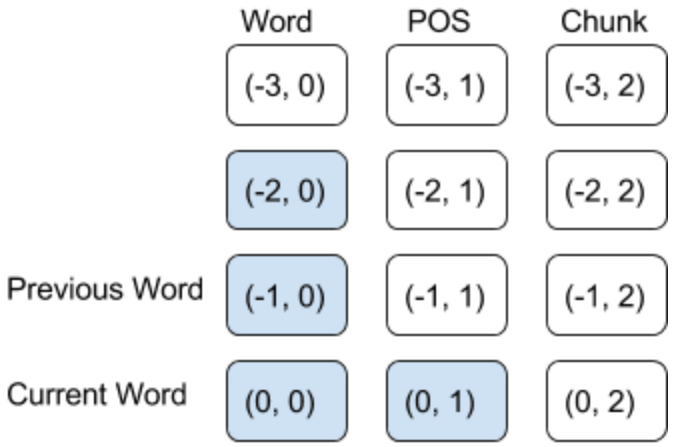
Machine learning methods learn from input data where the data consists of feature sets. In our project this sets are consisting of words and POS tags of previous lines. We have developed a mechanism that enables us to change set of features more dynamically.

The feature selection mechanism takes an array of tuples where each tuple consists of *"relative line position" and "feature of the line (word or POS tag)"*. For example,

- (-1, 0) => Previous word, where
- (-2, 1) => POS tag of the word that is 2 positions before from current word.

So at the end, using our "feature building mechanism" you can get following

feature set from “[(-2, 0), (-1, 0), (0, 0), (0, 1)]”:



The *featureSetBuilder* function is designed for this purpose and works by iterating line by line on given dataset and generating feature records like,

{

“f1”: Word at (-2, 0),

“f2”: Word at (-1, 0),

“f3”: Word at current location,

“f4”: POS Tag of current word

}

Alongside with feature sets, there are label sets too. Label sets are consisting from list of labels, e.g. [“I-NP, “B-VP”, “I-ADJP”, ….].

Classification algorithms are working on number sets, but in our case we have sets of texts. So we need a proper way to transform set of texts to set of numbers. This is where vectorization comes to help. Using vectorization we get a matrix where each row represents a line of data and each column represents a unique word. The general schema of the matrix can be shown like below.

	W1	W2	W3	W4	W5	W6	W7	....	Wn
R1	1	0	0	0	0	0	0	....	0
R2	0	0	1	1	0	0	0	....	0
R3	0	0	1	0	0	1	0	....	0
R4	0	0	0	0	0	1	0	....	1
R5	0	0	1	0	0	0	1	....	0
R6	0	1	0	0	1	0	0	....	0
:	:	:	:	:	:	:	:	....	:
Rn	1	0	0	1	0	0	0	....	0

But there is one big problem with the matrix shown above - it is too big. In fact, for given datasets, this matrix can't be fitted to the memory. Fortunately, solution for this problem is simple - store the matrix in sparse form (by passing *sparse=True* as an argument to the vectorizer class), that is to store only indices of 1's. This way it will take much less memory than previous one.

Training

There are many built-in classifiers in scikit library. Generally usage of classifiers is done by applying following steps(which are implemented in *trainer* function):

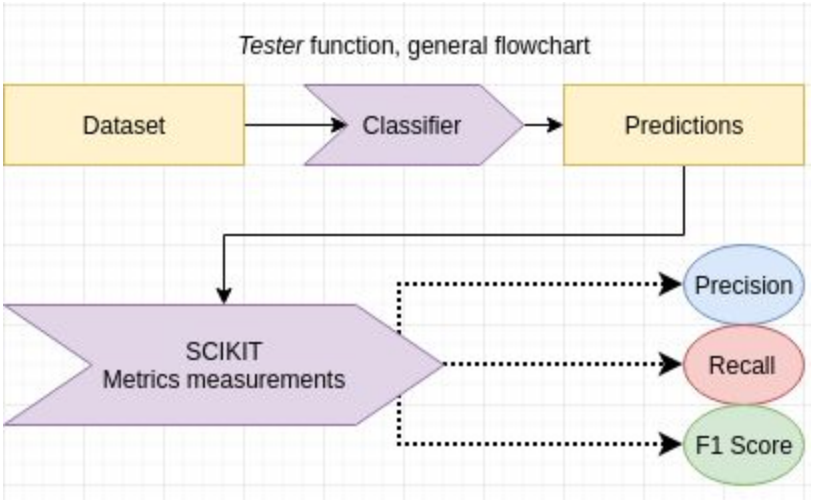
1. Instantiate a classifier by passing arguments that you want (all none passed arguments will have default values)
2. Train the classifier using *classifier.fit()* method.

After this steps classifier will be able to predict from new features. The *trainer* function was specifically designed for this purpose. We have implemented three classification methods in our project - Multinomial Naive Bayes, Linear Regression and Linear Support Vector Machine.

Testing and Scoring

Scoring is done by predicting features from test set and comparing the predictions against the actual labels (*testing* function is responsible for this part). Our

solution uses precision_recall_fscore_support function from sklearn.metrics package.



Execution

The modular approach of the project leads to simpler interface for writing the main code. In fact, training, testing and getting results can be written just in 6 lines of code as shown below,

```
1. featureSet = [(-1, 0), (-1, 1), (0, 0), (0, 1)]
2. train_features, train_labels = file_to_features_labels(FILE_TRAIN, featureSet)
3. test_features, test_labels = file_to_features_labels(FILE_TEST, featureSet)
4. classifier = MultinomialNB()
5. vectorizer, clf = trainer(train_features,train_labels, classifier)
6. results = tester(test_features, test_labels, vectorizer, classifier)
```

Of course for using above function you need to import them from *util.py* and *scikit*.

Batch execution - testing with different parameters

While learning about different classifiers of *scikit API* we have observed that there are number of parameters for each classifier. This parameters provide a way for tweaking the classifier for our needs. Although the default parameters were working good enough, we wondered about how different parameter combinations will affect the output of scores.

For this purpose, we have written batch testing functions for each classifier at

batch.py file. The general working principle of each function is a follows:

1. Define multiple variable sets that will change for each execution
2. Generate combination of them
3. Remove invalid combinations according to documentation of classifier
4. Sort parameters according for more readable output
5. Load feature/label sets from train/test datasets
6. Make tuples of classifiers and their parameters
7. Train and test on this classifiers
8. Return result scores

Above steps are working without any problem except long processing time. To reduce the processing time we have implemented a threaded execution functions. This functions take a list of dictionaries where each dictionary consists of an instance of initialized classifier and its parameter. Then each classifier is executed on a separate thread, where it is trained and tested. At the end all results will be collected from executed threads and returned.

Threaded batch execution model allows us to use all CPU-cores (in our case there are 8 virtual cores) and therefore reduce processing time. Also it is worse to mention that in threaded approach more memory is required but haven't observed a memory consumption above 3 - 3.5 GBs by our code (OS and other program usages are not included).

As a final part of this section, we want to inform that this kind of parameter iteration is not a best idea to find the optimal parameters. There are other good methods to find optimal values but for the scope of this project we think our approach is enough to see how parameter values affect the results.

Results & Analysis

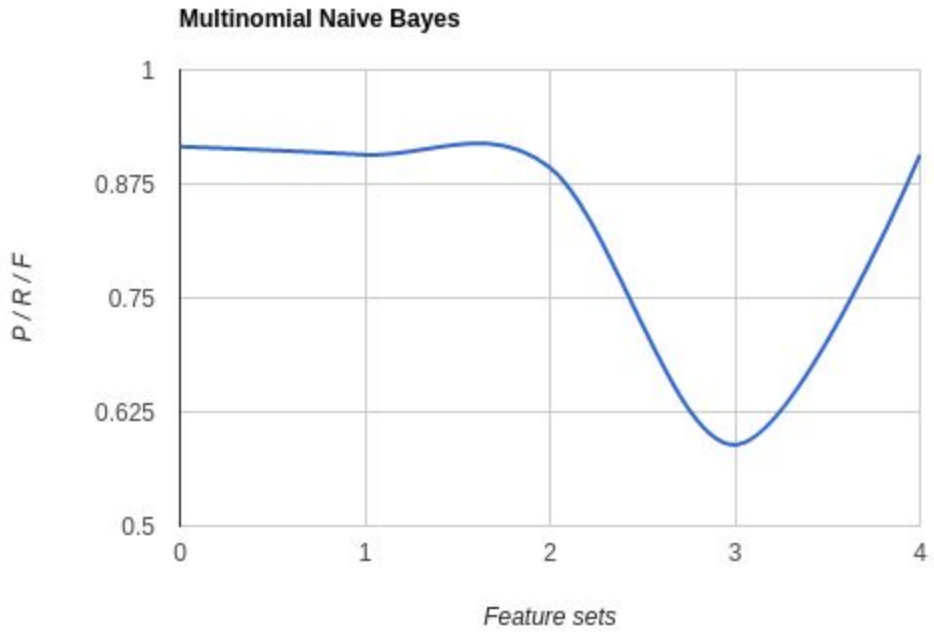
Feature sets

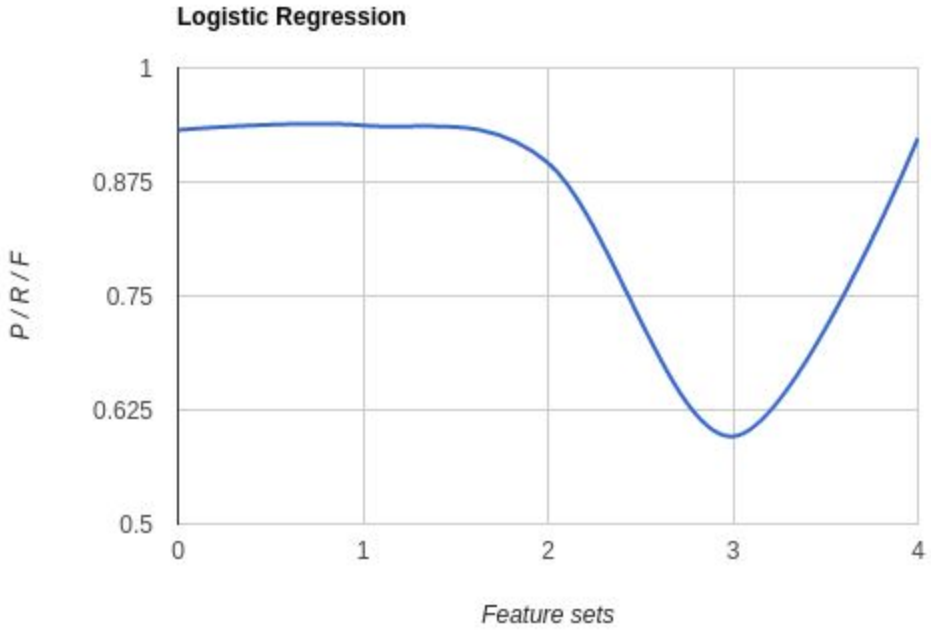
We have chosen five different patterns for feature sets. They are listed below:

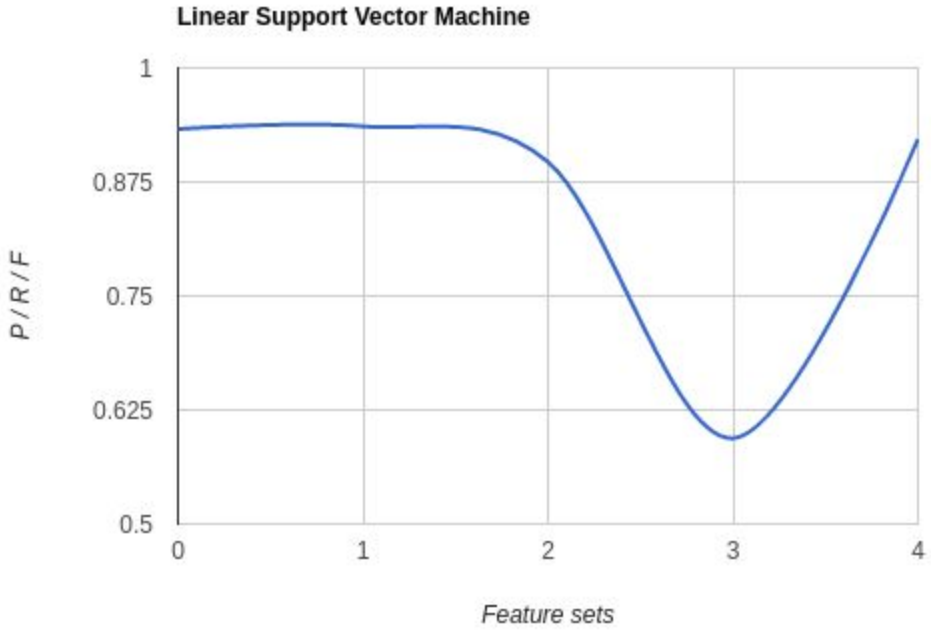
1. [prev token] [prev POS] [curr token] [curr POS]
2. [2 prev token] [2 prev POS] [prev token] [prev POS] [curr token] [curr POS]
3. [prev POS] [current POS]
4. [2 prev POS] [prev POS]
5. [2 prev POS] [prev POS] [curr token] [curr POS]

Run results with default parameter values

Classifier	Feature set #1	Feature set #2	Feature set #3	Feature set #4	Feature set #5
Multinomial Naive Bayes	0.916	0.907	0.893	0.589	0.907
Logistic Regression	0.932	0.937	0.896	0.596	0.923
Linear Support Vector Machine	0.933	0.936	0.897	0.594	0.922




Logistic Regression
1
0.875
0.75
0.625
0.5
P / R / F
0
1
2
3
4
Feature sets


Linear Support Vector Machine
1
0.875
0.75
0.625
0.5
P / R / F
0
1
2
3
4
Feature sets

From above results the worst results was encountered at 4th feature set and it was expected by us. We think, the main reason beyond that is the non-connectivity of features with current line. In fact there is an effect like we label previous line at current line.

The best results were acquired at Logistic Regression but difference with the "worst" classifier (Multinomial Naive Bayes) is only %2. This improvement may seem to be little, but for such applications it is significantly good improvement.

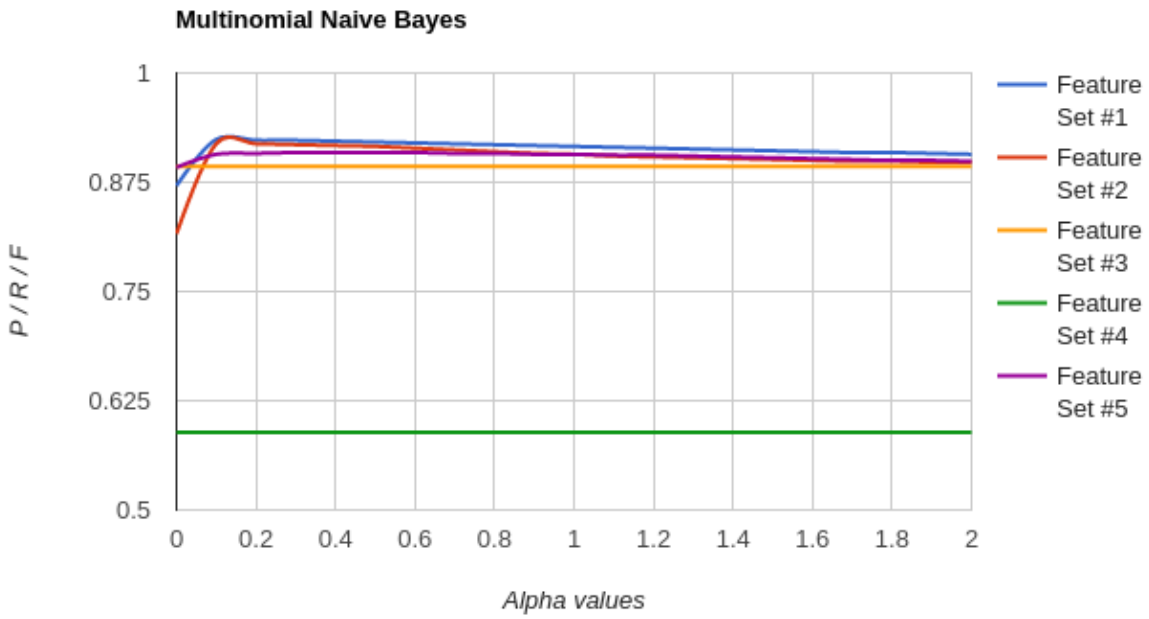
Another result we get from the table is the equality of results between Logistic Regression and Linear SVM. While running our experiments the Linear SVM was taking about 5x more time than Multinomial NB where Logistic Regression(Logit) was taking about 2x more time. This leads to conclusion that for given dataset, there is no need to spend computational powers on Linear SVM. Although Logit is taking more time, the improvement is enough to choose it.

Batch execution results

Before we give and analyse scores of classifiers with different parameter values, we would like to inform you that Precision, Recall and F1 scores of this classifiers we all equal to each other. We haven't used manual calculations and all values were calculated by functions provided from SCIKIT library.

In the below text in Precision, Recall and F1 scores are relating to each other.

Multinomial Naive Bayes



SCIKIT API provides Multinomial Naive Bayes classifier with a MultinomialNB class and it takes an *alpha* as argument. In the above graph we have given scores according to different *alpha* values (in range from 0 to 2) and feature sets. From above graph we can conclude that for feature sets #1 and #2 the best score was encountered at alpha=0.2 and feature set #5 continues to increase after alpha=0.2 but change is very little. Also scores of the feature sets #3 and #4 doesn't change with different alpha values. So in the last case, tweaking parameters in useless.

As a comparison with default parameters, it is obvious that by tweaking alpha we can get better results(0.916 vs 0.923) but improvement isn't enough to get closer to Logistic Regression.

Logistic Regression

Linear Regression or Logit is the second classifier that we have tried and the resulting scores were impressive against other classification methods. Using logit we can get ~ 2% improvement. Unfortunately this result doesn't come without any cost - the training time is increasing about two times. But as training time of Multinomial Naive Bayes was very short, increase in time doesn't matter so much and 2% improvement is pretty good result for our purpose (in the next section you can find more information about advantages of Logit).

LinearRegression class takes a large number of parameters but we have done our experiments on six of them - penalty, dual, C, max_iter, solver and multi_class. After generating combinations of this parameters there were 198 different combinations and as was mentioned in previous sections there are 5 feature sets requiring to make 198*5 = 990 different experiments. Fortunately we were able to make this experiments and get results.

Due to large number of results we aren't able to show graphs of them but in this and following paragraphs we will try to discuss about results. For feature set #1 we have got 0.934 precision value with C = 5, max_iter=10 and solver = "newton-cg". Same results was obtained at C=1 with solver="newton-cg" and max_iter=20. The same parameters were resulted in best scores of other features. Also we have observed that other solvers like "sag" and "lbfgs" were able to give same scores in a similar parameters. So little C and max_iter values lead to more good scores.

From above paragraph you can observe that there were changes but they were small when comparing with default parameters. In fact we were able to get best scores with custom parameters that are higher from default parameters by only 0.002 - 0.005. As a result we have concluded that for given datasets there is no need for using custom parameters.

And the final observation was the bad performance feature set #3 (relatively bad) and feature set #4 (very very bad). But this feature sets are doing no good at the other classifiers too. So it is not related with Logit but with selections of features. Other feature sets perform nearly equal performance for all of the classifiers.

Linear Support Vector Machine

Before we talk about Linear SVM we want to explain why we have chosen it instead of ordinary SVM. The reason of this selection is the following statement from API documentation,

> The implementation is based on libsvm. The fit time complexity is more than quadratic with the number of samples which makes it hard to scale to dataset with more than a couple of 10000 samples [source: SCIKIT, sklearn.svm.SVC]

Unfortunately we have implemented ordinary SVM at the beginning stages of the project which led to execution time more than 5 minutes. On the other hand, Linear SVM can handle large number of samples easily.

Linear SVM is used by passing appropriate arguments to LinearSVC class. This class takes many parameters but we have done our experiment based on 6 of them - C, loss, penalty, dual, multi_class and fit_intercept. Below you can find comparisons of different C values while other parameters remaining constant.

The graph at the end of this section shows different scores for different C values (which shows how “strict” is our model, used to avoid overfitting). It is obvious that scores remain constant for large C values and best score occurs when C=1. Another observation is the scores of feature set #4 and #5. This feature sets have done very bad work (especially #4) as it was in previous classifiers.

When tracing all results it is easily seen that best scores occur under following parameter values (which lead to the same results as default parameters):

- Loss = squared_hinge
- C = 1
- Fit_intercept = False
- Penalty = Level 2
- Multi_class = OVR
- Dual = False

You can find all results of Linear SVM experiments in project folder.