

Saher W Yakoub

CSC 594 – Text Mining

Midterm project – Sentiment Analysis & Text Classification

Table of contents:

System Version 2.0

Introduction	pg 2
Knowing The System	pg 3
Parameters Page	pg 4
Single / Full Test	pg 6
Future Enhancement	pg 7

System Version 1.0

Introduction	pg 8
Data Used	pg 9
Phase 1 – User Input	pg 10
Phase 2 – Text Pre-processing	pg 11
Common between Classifiers	pg 15
Phase 3a – NaiveBays Classifier	pg 17
Phase 3b – DecisionTree Classifier	pg 19
Phase 3c – MaxEnt Classifier	pg 21
Phase 4 – Maximum Vote	pg 23
Comparison & Conclusion	pg 25

System Version 2.0

What is new?

- Converting **V1.0** to a standalone web framework with HTML GUI, using Flask
- Building a **Full Responsive Mobile Accessible** HTML / CSS3 / JS web framework UI.
- Adding a layer of customization for the system's parameters, to be customized due to the user preferences.
- Enhancement on the **Full Output** interface, and making it more readable and fun.
- Enhancement on the **Single Test** interface, making it more customizable and easy to use.

System prerequisites?

- Pre-installed Python
- Pre-installed **Flask** (Installation on Windows might be tricky, so please follow the provided link for installation <http://flask.readthedocs.org/en/0.3.1/installation/#easy-install-on-windows>)
- Pre-installed NLTK module, as the system uses it **along with its movie_review corpus.**

How to run?

- Import the whole project **SATC** into your favorite Python IDE.

- Run the file **routes.py**

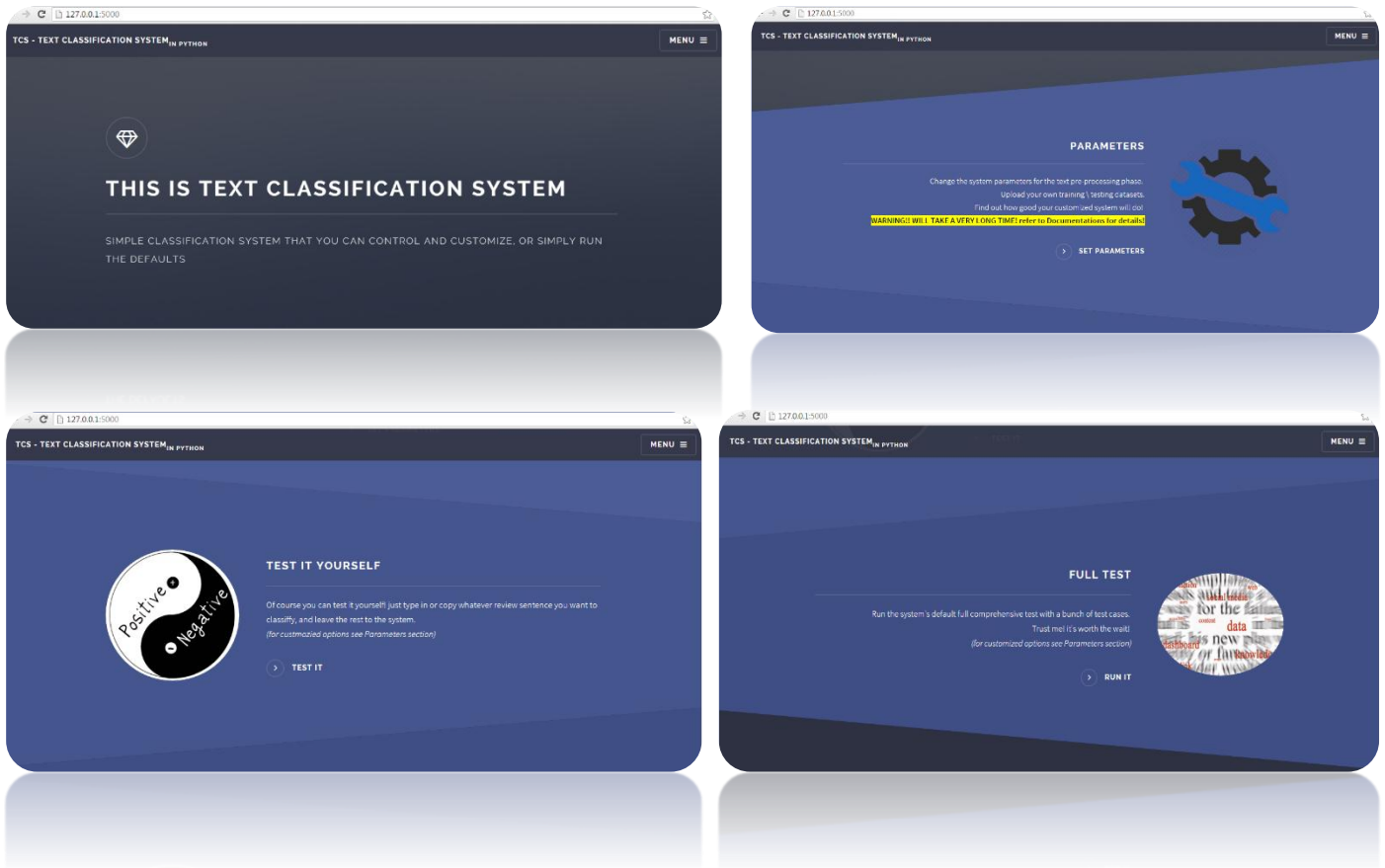
```
C:\Anaconda\python.exe D:/MASTER/PycharmProjects/SATC/TCS/routes.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
```

- Open your favorite browser, and open the provided portal to your application

- **Enjoy**

Knowing the System?

After simply entering **127.0.0.1:5000** in the browser, you'll be redirected by the application to **index.html** which will have **four main section** (Parameters – Single Test – Full Test – Documentation - About).



Each section of the above does exactly what it says....

- **Parameters:** Allows the user to control the system's configuration for training and testing the classifiers.
- **Test It Yourself:** Allows the user to run a one statement test, and choose which classifier to run the test one.
- **Full Test:** Allows the user to run a test on **500 testing dataset**, and output a full analysis for Accuracy, Recall, and Precision.

NOW, we will have closer look at each of the above section in details.

NOTE, the system's classifiers technical details, are already discussed in **V1.0**, and will not be addressed here.

Parameters:

One of the most, **if not the hardest** modules of the system.

Allowing the user to choosing between a four **pre-defined** combinations for the system configuration.

	Option Default	Option 1	Option 2	Option 3
Remove Punctuation		✓	✓	✓
Lower Case		✓		
Stop words Removal	✓	✓	✓	✓
POS Tagging			✓	✓
Lemmatization		✓	✓	✓
Bigram Finding	✓			
Word Frequency Count	✓	✓	✓	
Training Time	2-3 M	N\A	5 H	N\A

After choosing which options to train with, and hitting **TRAIN**, the page will be loading **unchanged** until the classifiers are trained and returned, then you'll be redirected to a Page to confirm that your training was done successfully.

Some comments about the parameters:

- **Remove Punctuation:** using the Python method **translate** with **string.punctuation** argument for the set of words to be removed.
- **Lower Case:** using the simple string method **.lower()**
- **Stop Words Removal:** Using NLTK's stopwords dictionary.
- **POS Tagging:** Keeping only the following tags (JJ – JJR – JJS – RB – RBR - RBS).
- **Lemmatization:** there are two versions of this method, one that comes with POS tagging, and the other is the default one. **(For more accurate result, POS & lemmatization should be used together).**
- **Bigram Finding:** Using the **bigramCollectionFinder** from NLTK collection module.
- **Word Frequency Count:** Discussed in details in **v1.0**.

Important notice

The following logic is followed with the customized parameters training:

- For each entry in the training / testing dataset do:
 - filter the entry as per the option chosen, and then put it in the training / testing set
 - call classifier train method on the filtered training dataset.

On an i5 laptop this method took 5 hours!!!

- Please, be ready to go back and forth between the browser and your IDE, as the training progress will be shown in the IDE NOT the browser. I haven't yet figured out a way to show a results from a Python method before it completely returns.

- After successfully training your classifiers, you can go back and run a **full test** or a **single test** as usual. There is still a problem with passing the configuration to the testing phase, so the results might be a little misleading

- The parameters are there, to prove that the logic behind it DOES WORK, but there is still so much work to do to reach the full customization support for the system.

- I you want to test that the parameters customization works! You're more than welcome to try it, but I still recommend going for the default option when testing

TCS - TEXT CLASSIFICATION SYSTEM

IN PYTHON

MENU

CONFIGURE YOUR OWN PARAMETERS

WARNING!! WILL TAKE A VERY LONG TIME! refer to Documentations for details!

☒ OPTION Default

BEST PERFORMANCE

LEAST TRAINING TIME

☐ OPTION 1

Punctuation Removal

Lower Case

Stop Words Removal

Lemmatization

Word Frequency

☐ OPTION 2

Punctuation Removal

Stop Words Removal

POS Tagging

Lemmatization

Word Frequency

☐ OPTION 3

Punctuation Removal

Stop Words Removal

POS Tagging

Lemmatization

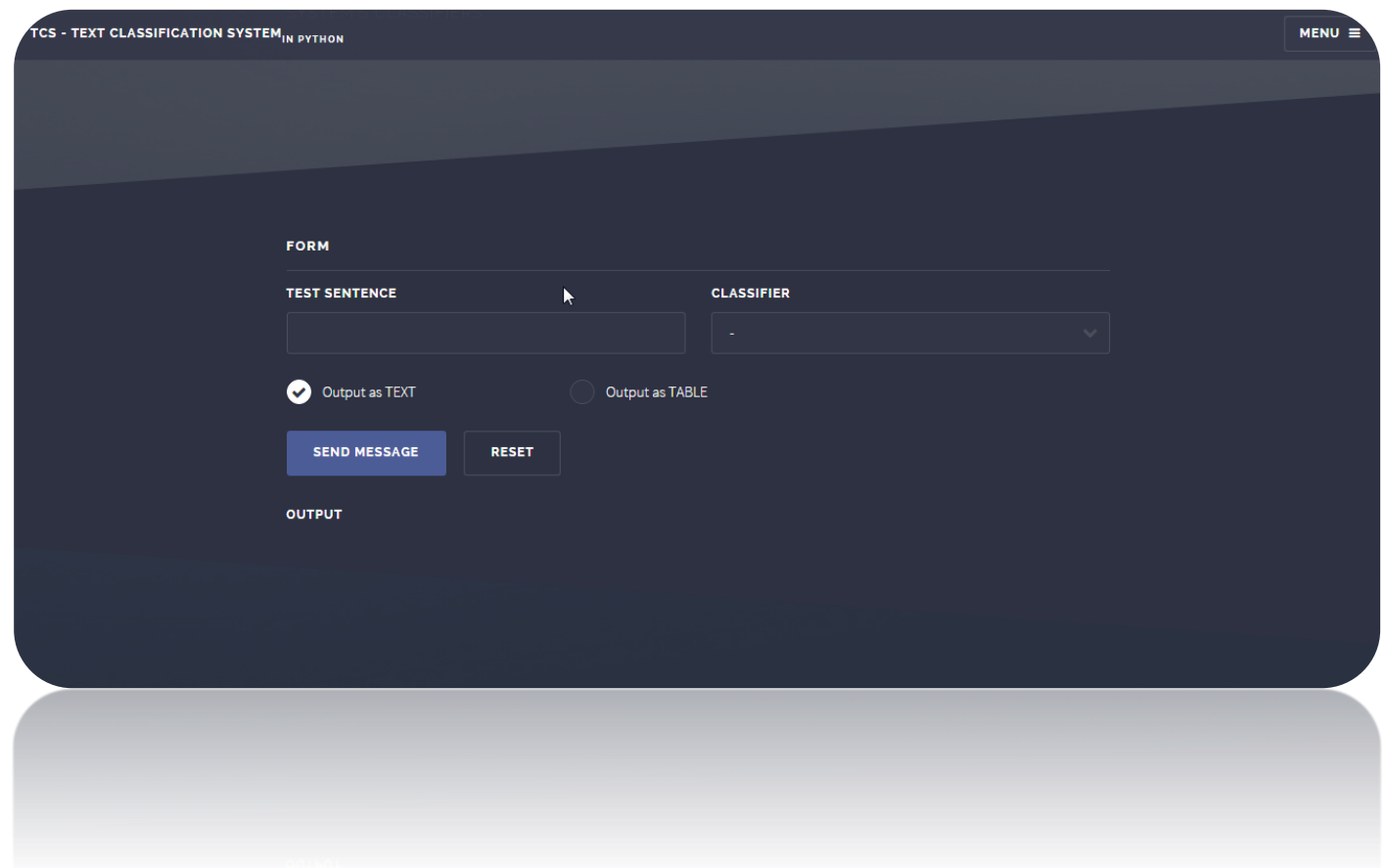
TRAIN

RESET

Single Test:

Allowing the user to run the system on a single entry test, specifying the **test entry**, **classifier to be used**, and **the output format**.

As simple as it can gets, this phase is more or less the same as **option 1** in **v1.0** where you can enter a test entry, here you can also choose which combination of classifiers you want the system to use.

The image shows a web application interface for a Text Classification System. The header bar is dark blue with the text 'TCS - TEXT CLASSIFICATION SYSTEM' and 'IN PYTHON' on the left, and a 'MENU' button with a hamburger icon on the right. The main content area has a dark blue background. Under the heading 'FORM', there are two input fields: 'TEST SENTENCE' and 'CLASSIFIER'. Below these are two radio buttons: 'Output as TEXT' (which is selected) and 'Output as TABLE'. At the bottom of the form are two buttons: 'SEND MESSAGE' and 'RESET'. Below the form is an 'OUTPUT' section, which is currently empty. The entire interface is reflected in a light gray shadow below it.

Full Test:

Again, does exactly as it says, running the full test analysis on the system's classifiers.

Outputting the result in both **Text** and **Tabular** views, which the user can toggle between.

However, for both **full & single tests** be ready to go back and forth between the browser and your IDE to follow the classifier progress, as I haven't figured out a way to display that on the browser yet. **YOUR CURRENT PAGE WILL NOT BE AFFECTED UNLESS THE WORK IS DONE IN THE BACKGROUND.**

Future Enhancements:

Of course this is not an optimal final system, for such a big concept! And I REALLY looking forward to continue working in this system, as I had so much fun starting it and getting the progress up to that level.

What can be done?

- Figure a way to display Python progress on the browser, instead of loading endlessly.
- Display **at least** a loading screen at the browser, instead of just a little tab loading icon.
- Achieving an optimal **deploying** mechanism, which allows the user to only focus in the system's UI not going back and forth between the browser and IDE.
- Achieving a full **transparency** and **optimization level** for the system, where the configured parameters can be used in both training and testing phases.
- Allowing the user to upload **any corpus** to be used with the classifiers.
- Finding out **WHY** it is taking so much time to train the data against the configured parameters, and **where** did this go wrong?
- Getting rid of **redundancy** in the system's logic, as for now there are **lots** of steps that are being executed more than once.

Acknowledgment:

I know this is not supposed to be part of the documentation, but I'll just put it here just in case my professor is actually reading a 26 pages documentation, and for my future reference.

* I had **SO MUCH FUN** implementing **V1.0** of the system, and I was so proud to get such a positive feedback for it from the professor.

* I had **EVEN MORE FUN** implementing **V2.0** of it! I mean the amount of stuff that I learned in this course and this system (from both ends **Text Mining & Software Development**) is enormous.

* I'm just **Thankful**, and really hope that my professor like it as much as I do.

* **I promise** to keep working on it, and get to higher stable, reliable, optimal versions

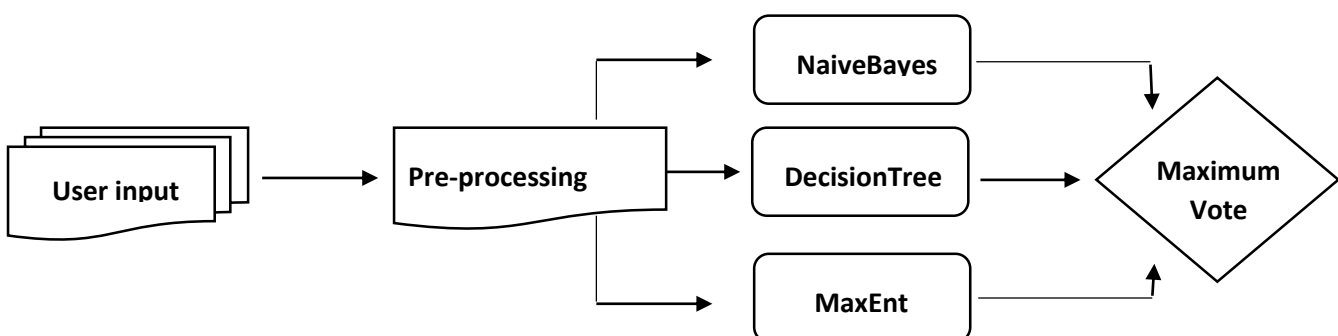
*** **END OF V2.0** ***

Project outline:

- The main purpose of this project is to write a machine learning program that is able to determine **positive** and **negative** statements, according to some kind of classification technique. And see how accurate it will end up to be?
- Another goal for this project is to apply some different **text pre-processing** techniques on the training & testing sets we have, and see how this will affect our learning module and our accuracy ratio as an overall.

How it works?

- **Phase 1:** The user chooses the preferred input method (writing – feeding a file – running test cases)
- **Phase 2:** a text pre-processing filtering is done to raw data before continuing.
- **Phase 3a:** the filtered corpus is passed to a **NaiveBayes classifier** to get some results.
- **Phase 3b:** the same corpus is passed to a **DecisionTree classifier** to get some results.
- **Phase 3c:** Yet another copy from the corpus is passed to a **MaxEnt classifier**.
- **Phase 4:** a **MaxVote classifier** gathers the three classifiers' votes and make a decision according to the voting results.



Data Used (corpus)

Total of 2,000 movie reviews which are 7.42 MB.

Positive reviews are 1,000

Negative reviews are 1,000

Constructed as follows:

```
movie_reviews
|-- neg
|   |-- cv000_XXXXXX.txt
|   `-- cv999_XXXXXX.txt
`-- pos
    |-- cv000_XXXXXX.txt
    `-- cv999_XXXXXX.txt
```

Each file contains a single review.

Note 1: These files will be used for training & testing the module.

Training set will be 1,500 file from each category (pos / neg)

Testing set will be 500 files from each category (pos / neg)

Note 2:

Those files are already built-in inside *nltk's corpus module*, and can be imported by *Python import statement*. The data will be available for download with the source code and the documentation. **BUT** for the sake of convenience it will directly imported from *nltk corpus data* in the source code.

Phase 1 – User input *Intro.py*

What is it?

The program allows the user to choose from many input methods provided (figure 1).

User can:

- Manually insert the review the needs to be classified.
- Give a file path to be opened and examined and output the results.
- Run the default **comprehensive** test case to test all the module.
- Exit the program.

Figure 1

```
C:\Anaconda\python.exe
Select an action...
1) Enter a test sentence
2) Evaluate a file (demo)
3) Run test cases
4) Exit
```

How it works?

- By running **"Intro.py"** the user will be prompted with the options shown in Figure 1.
- The main method just captures the input from the user.
- Pass the value of the input to a start function, who has a Python Dict that contains corresponding functions for each possible user entry.
- Functions do as expected...

Either to call the Classifiers' Module and evaluate a user sentence,

Take a file path to open it and evaluate all sentences found inside,

Run the comprehensive test case,

Or exit the program.

Code snippet:

Figure 2

```
def f1():
    """
    For choice number 1 - user input review
    :return: each classifier's classify to the user input
    """
    print "Enter your review: "
    choice = raw_input()
    nb = Classifier_NB.load_classifier()
    dt = Classifier_DT.load_classifier()
    me = Classifier_ME.load_classifier()
    mv = Classifier_MV.MaxVoteClassifier(nb,dt,me)

    test = Featx.bag_of_non_stop_words(choice.split())
    print "NaiveBayes      --> " + str(nb.classify(test))
    print "DecisionTree   --> " + str(dt.classify(test))
    print "maximum entropy --> " + str(me.classify(test))
    print "Maximum Vote    --> " + str(mv.classify(test))
    print "\n"
    return main()
```

Figure 3

```
def main():
    """
    Main method
    :return: 0
    """
    while True:
        print("Select an action...")
        print("1) Enter a test sentence")
        print("2) Evaluate a file (demo)")
        print("3) Run test cases")
        print("4) Exit")
        choice = raw_input()
        if re.match('[1-4]', choice):
            # print("Your choice was " + choice)
            start(choice)
            break
        else:
            print("Sorry!! Wrong input...\n")
```

Phase 2 – Text Pre-processing *Featx.py*

What is it?

The idea behind text pre-processing, is transforming the normal unstructured text sets into a feature set that is usable by the classifier. NLTK classifier expects a **Dict**.

The *bag_of_words* module is essentially the piece of code that produce that Dict in a shape of {key: value, word: True}, and the returned Dict is a bag because the words are not in order. All that matters is that the word occurred at least once in the sentence (Figure 4).

In the default *bag_of_words* module, all words treated equally, but that is not always a good idea, and that's where the rest of pre=processing modules come into play.

Figure 4

```
The beautiful bird is singing on the tree
{'beautiful': True, 'on': True, 'is': True, 'tree': True, 'the': True, 'singing': True, 'bird': True}
```

How it works?

This project only does two of the text pre-processing techniques out of so many, **filtering stop words, and including significant bigrams**. It doesn't do **part of speech tagging (POS), stemming, or lemmatization** for example.

Filtering stop words: As we know not all words should be treated equally, as they are not of the same importance. Some words are common to be meaningless, so we are going to just exclude them from the tested sentence and will see if that make a difference to our test cases. (Discussed later)

bag_of_non_stopwords, and *bag_of_words_not_on_set* are the two modules responsible for filtering the stop words for us.

Bag_of_non_stop_words returns to lists, 1st is a list of words in the sentence, and 2nd is the list of stop words in a given language (*default to English*). Then the two lists are passed to *bag_of_words_not_in_set* to remove the stop words from the original list to produce a filtered list that is being passed to *bag_of_words* module to return the Dict (Figure 5).

Figure 5

```
The beautiful bird is singing on the tree
{'beautiful': True, 'tree': True, 'singing': True, 'bird': True}
```

✚ As explained by Figure 4 & Figure 5, Calling *bag_of_words*, and *bag_of_non_stop_words* on the same sentence produced different outputs.

Including significant Bigrams: In addition to single words, it often helps to

include significant bigrams using *theBigramCollocationFinder*.

Figure 6

As significant bigrams are less common than most individual words, including them in the Bag of Words can help the classifier make better decisions.

bag_of_bigrams_words will return a dict of all words along with the 200 most significant bigrams (Figure 6).

```
The beautiful bird is singing on the tree
beautiful True
(('bird', 'is'), True) True
(('the', 'beautiful'), True) True
(('is', 'singing'), True) True
(('on', 'the'), True) True
(('singing', 'on'), True) True
tree True
(('the', 'tree'), True) True
(('beautiful', 'bird'), True) True
singing True
bird True
```

Calculating high information words:

A **high information word** is a word that is strongly biased towards a single classification label. These are the kinds of words we see by calling the classifier method *show_most_informative_features* for both the *NaiveBayesClassifier* and the *MaxentClassifier*.

The **low information words** are words that are common to all labels. It may be counter-intuitive, but eliminating these words from the training data can actually improve accuracy, precision, and recall. The reason this works is that using only high information words reduces the noise and confusion of a classifier's internal model.

If all the words/features are highly biased one way or the other, it's much easier for the classifier to make a correct guess.

Somewhat surprisingly, the top words are different for both classifiers. This discrepancy is due to how each classifier calculates the significance of each feature, and it's actually beneficial to have these different methods as they can be combined to improve accuracy.

The way this function works:

The *high_information_words* function starts by counting the *frequency* of every word, as well as the *conditional frequency* for each word within each label. This is why we need the words to be labelled (input is a list of 2-tuples in the form of [(label, words)]), so we know how often each word occurs in each label.

Once we have this *FreqDist* and *ConditionalFreqDist*, we can score each word on a per-label basis. The *defaultscore_fn* is *nlTK.metrics.BigramAssocMeasures.chi_sq*, which calculates the *chi-square* score for each word using the following parameters:

1. *n_{ji}*: The frequency of the word in the label.
2. *n_{jx}*: The total frequency of the word across all labels.
3. *n_{xi}*: The total frequency of all words that occurred in the label.
4. *n_{xx}*: The total frequency for all words in all labels.

The simplest way to think about these numbers is that the closer n_{ii} is to n_{ix} , the higher the score. Or, the more often a word occurs in a label, relative to its overall occurrence, the higher the score.

Once we have the scores for each word in each label, we can filter out all words whose score is below the *min_score*. We keep the words that meet or exceed the threshold, and return all high scoring words in each label.

NOW as we have the high information words, we use the feature detector function *bag_of_words_in_set*, which will let us filter out all low information words.

For the purpose of a full comprehensive **test cases** we have multiple copies of this function, one of them filters the produced list (*high_informative* & *all_words*) from stop words, and the other don't. Just to see if that will make any improvements.

Code Snippet:

Figure 7

```
def high_information_words(labeled_words, score_fn=BigramAssocMeasures.chi_sq, min_score=5):
    """
    To eliminate low information feature words for set of words for EFFICIENCY
    :param labeled_words: list of 2 tuples [(label, words)]
        label -> is a classification label (pos / neg)
        words -> is a list of words that occur under that label
    :param score_fn: a scoring function to measure how informative that word is
    :param min_score: the minimum score for a word to be included as MOST INFORMATIVE WORD
    :return: a set of high informative words
    """
    word_freq = FreqDist()
    labeled_word_freq = ConditionalFreqDist()

    for label, words in labeled_words:
        for word in words:
            word_freq[word] += 1
            labeled_word_freq[label][word] += 1
    n_xx = labeled_word_freq.N()
    high_info_words = set()

    for label in labeled_word_freq.conditions():
        n_xi = labeled_word_freq[label].N()
        word_scores = collections.defaultdict(int)

        for word, n_ii in labeled_word_freq[label].iteritems():
            n_ix = word_freq[word]
            score = score_fn(n_ii, (n_ix, n_xi), n_xx)
            word_scores[word] = score

        bestwords = [word for word, score in word_scores.iteritems() if score >= min_score]
        high_info_words |= set(bestwords)

    return high_info_words
```

High Informative
↓
Pass to filter low
informative

Figure 8

```
def bag_of_words_in_set_sw(words, goodwords):
    """
    To get a Dict of the most informative feature words INCLUDING stopwords
    :param words: set of words
    :param goodwords: set of high informative words
    :return: a combination of high informative words + words and pass it to bag_of_words
    """
    return bag_of_words(set(words) & set(goodwords))

def bag_of_words_in_set_nsw(words, goodwords):
    """
    To get a Dict of the most informative feature words EXCLUDING stopwords
    :param words: set of words
    :param goodwords: set of high informative words
    :return: a combination of high informative words + words and pass it to bag_of_words
    """
    return bag_of_non_stop_words(set(words) & set(goodwords))
```

The Union

Pass to find
Bigrams

Figure 9

```
def bag_of_bigram_words(words, score_fn=BigramAssocMeasures.phi_sq, n=200):
    """
    to get a list of coupled words from a list of words
    :param words: list of words
    :param score_fn: function to be used in creating the Bigram
    :param n: the number of the most significant bigrams needed
    :return: list of bigram words and pass it to bag_of_words to get a Dict
    """
    bigram_finder = BigramCollocationFinder.from_words(words)
    bigrams = bigram_finder.nbest(score_fn, n)
    filtered_bigram = bag_of_non_stop_words(bigrams).items()
    words = [i for i in words]
    return bag_of_non_stop_words(words + filtered_bigram)
```

Collect Bigrams

Pass to find
stopwords

Figure 10

```
def bag_of_non_stop_words(words, sw='english'):
    """
    to filter all english stop words
    :param words: list of words
    :param sw: language of the stopwords
    :return: list of words + list of stopwords and pass for filtering
    """
    badwords = stopwords.words(sw)
    return bag_of_words_not_in_set(words, badwords)
```

Stopwords list
+ all words

Pass to filter

Figure 11

```
def bag_of_words_not_in_set(words, badwords):
    """
    to make a list of FILTERED words to be used
    :param words: list of words
    :param badwords: words need to be extracted
    :return: list of words to be USED and pass it to bag_of_words
    """
    return bag_of_words(set(words) - set(badwords))
```

Filtered words

Pass to get a
Dict

Figure 12

```
def bag_of_words(words):
    """
    to convert a list of words into a boolean Dict
    :param words: any list of words
    :return: Dict [(word, True)]
    """
    return dict([(word, True) for word in words])
```

Dict of filtered words

Commons between classifiers:

- label features from corpus:

As discussed in the **Data Used** section, we are going to use the *movie_reviews* dataset for training and testing this classifier. The classification we will be doing will be in a form of *sentiment analysis*. If the classifier returns *POS* then the text expresses *positive sentiment*, if the classifier returns *NEG* then the text express *negative sentiment*.

For training we need to create a list of labeled feature sets in form of *[(featureset, label)]* where *featureset* is a Dict, and *label* is the known class label for the *featureset*. The *label_feats_from_corpus* function takes a corpus, such as *movie_reviews*, and a *feature_detector* function, which defaults to *bag_of_words*. It then constructs and returns a mapping of the form *{label: [featureset]}*. We can use this mapping to create a list of labeled *training instances* and *testing instances*.

Figure 13

```
def label_feat_from_corpus(corpus, feature_detector=Featx.bag_of_words):
    """
    To create a list of labeled feature set [(featureset, label)]
    featureset -> is a Dict
    label -> the known class for the featureset (pos / neg)
    :param corpus: any kind of corpus (movie reviews)
    :param feature_detector: a function to return a Dict (Default to bag_of_words)
    :return: a mapping set of (label: [featureset])
    """
    label_feats = collections.defaultdict(list)
    for label in corpus.categories():
        for fileid in corpus.fileids(categories=[label]):
            feats = feature_detector(corpus.words(fileids=[fileid]))
            label_feats[label].append(feats)
    return label_feats

def split_label_feats(lfeats, split=0.75):
    """
    To split that labeled mapped set to training & testing sets
    :param lfeats: mapping set returned from the label_feat_from_corpus
    :param split: the ratio to split
    :return: two sets for training & testing
    """
    trainset = []
    testset = []
    for label, feats in lfeats.iteritems():
        cutoff = int(len(feats) * split)
        trainset.extend([(feat, label) for feat in feats[:cutoff]])
        testset.extend([(feat, label) for feat in feats[cutoff:]])
    return trainset, testset
```

- Split labeled features:

Once we can get a mapping of *label: feature sets*, we want to construct a list of labeled *training instances* and *testing instances*. The function *split_label_feats* takes a mapping returned from *label_feats_from_corpus* and splits each list of feature sets into labeled *training* and *testing instances*.

- Accuracy:

Classifier's accuracy is simply determined by calling the *nlTK's built-in accuracy* function passing a trained classifier plus a test set constructed previously. But for the sake of easiness, we decided to put it in a callable function *Classifier_acc*.

Figure 14

```
def Classifier_acc(C, test_set):
    """
    to determine the accuracy of a classifier on a given test set
    :param C: Classifier
    :param test_set: test set
    :return: Accuracy percentage
    """
    return nltk.classify.accuracy(C, test_set)
```

- Run, load, and save:

For the sake of performance and simplicity, Classifiers can be **saved** as a trained instances, and **loaded** whenever needed by the *pickle* module, and that's what *run*, *save_classifier*, and *load_classifier* methods are for.

- Precision and Recall:

Last thing to be done to a classifier, is **testing the precision and recall** for the classifier.

To understand those two, we need to have a look on the following:

False positive: happens when a text gets classified with a label that it shouldn't have.

False negative: happens when a text doesn't get labeled that it should have.

True positive: happens when a text gets labeled as POS and it should be that way.

True negative: happens when a text get labeled as NEG and it should be that way

NOTE...in our case, we're looking into binary classification, so False positive & False negative happens at the same time.

NOW...Precision is the lack of *false positives* and **recall** is the lack of *false negatives*. The more precise a classifier is, the lower the recall.

Precision_recall function, is doing just that. It takes a trained classifier, and a testing set to build two sets for each label.

Then we build a *reference set* & *test set*. The first set contains the COREECT labels, and the second set contains the labels guessed by the classifier.

Passing that to *nlTK.metrics* we will be able to calculate the *Precision and recall* percentage for a given classifier.

Figure 15

```
def precision_recall(C, test_set):
    """
    :param C: trained classifier
    :param test_set: testing set
    :return: two Dict 1st holds the precision for each label
            2nd holds the recall for each label
    """
    refsets = collections.defaultdict(set)
    testsets = collections.defaultdict(set)

    for i, (feats, label) in enumerate(test_set):
        refsets[label].add(i)
        observed = C.classify(feats)
        testsets[observed].add(i)

    precisions = {}
    recalls = {}

    for label in C.labels():
        precisions[label] = nltk.precision(refsets[label], testsets[label])
        recalls[label] = nltk.recall(refsets[label], testsets[label])
    return precisions, recalls
```


Phase 3a – NaiveBayes classifier *Classifier_NB.py*

What is it?

Classifiers takes dictionaries as an input, label tokens with category labels (or *class labels*). Typically, labels are represented with strings (such as "health", "sports", "positive", or "negative").

In NLTK, classifiers are defined using classes that implement *theClassifyI* interface. And it defines several classifier classes:

- *ConditionalExponentialClassifier*
- *DecisionTreeClassifier* (**Used**)
- *MaxentClassifier* (**Used**)
- *NaiveBayesClassifier* (**Used**)
- *WekaClassifier*

Classifiers are typically created by training them on a training corpus (See Data Used).

How it works:

NaiveBayesClassifier uses **Bayes Theorem** to predict the probability that a given feature set belongs to a particular category.

$$\underline{P(\text{label} \mid \text{features}) = P(\text{label}) * P(\text{features} \mid \text{label}) / P(\text{features})}$$

- **P (label):** is the prior probability of the label occurring, which is the same as the likelihood that a random feature set will have the label. This is based on the number of training instances with the label compared to the total number of training instances. For example, if 60/100 training instances have the label, the prior probability of the label is 60 percent.
- **P (feature | label):** is the prior probability of a given feature set being classified as that label. This is based on which features have occurred with each label in the training data.
- **P (feature):** is the prior probability of a given feature set occurring. This is the likelihood of a random feature set being the same as the given feature set, and is based on the observed feature sets in the training data. For example, if the given feature set occurs twice in 100 training instances, the prior probability is 2 percent.
- **P (label | features):** tells us the probability that the given features should have that label. If this value is high, then we can be reasonably confident that the label is correct for the given features.

Training the classifier is by a simple line of code

```
nb_classifier = NaiveBayesClassifier.train(train_feats)
```

Testing the classifier is by a simple line of code (Figure 16).

```
nb_classifier.classify(test_sentence)
```

Calculating the accuracy for the classifier is by a simple line of code (Figure 16). Figure 16

```
Classifier_acc (nb_classifier, test_feats)
```

```
This movie is so good
pos
Classifier Accuracy: 0.728
```

Functions included:

- ✚ `Label_feats_from_corpus` -> see [Commons between classifiers](#)
- ✚ `split_label_feats` -> see [Commons between classifiers](#)
- ✚ `Classifier_acc` -> see [Commons between classifiers](#)
- ✚ `Precision_recall` -> see [Commons between classifiers](#)
- ✚ `run` -> see [Commons between classifiers](#)
- ✚ `save_classifier` -> see [Commons between classifiers](#)
- ✚ `load_classifier` -> see [Commons between classifiers](#)

Code snippet:

Figure 17

```
feats = label_feat_from_corpus(movie_reviews)
training, testing = split_label_feats(feats)
# print "Running Classifier"
# run()
# print "Classifier trained"
# print "Loading Classifier"
C = load_classifier()
test_sent = "This movie is so good"
print test_sent
print C.classify(Featx.bag_of_words(test_sent))
# print C.show_most_informative_features(5)
print "Classifier Accuracy: " + str(Classifier_acc(C, testing))
np_p, np_r = precision_recall(C, testing)
print "Precision Positive" + str(np_p['pos'])
print "Precision Negative" + str(np_p['neg'])
print "Recall Positive" + str(np_r['pos'])
print "Recall Negative" + str(np_r['neg'])
```

Output

Figure 18

```
This movie is so good
pos
Classifier Accuracy: 0.728
Precision Positive0.651595744681
Precision Negative0.959677419355
Recall Positive0.98
Recall Negative0.476

Process finished with exit code 0
```

Phase 3b – DecisionTree classifier *Classifier_DT.py*

What is it?

Decision trees are a decision support models that uses a tree-like graph for decisions and their possible consequences. In its simplest form decision trees are similar to *if-elif-else statements*.

How it works?

The *DecisionTreeClassifier*, like the *NaiveBayesClassifier*, is also an instance of *ClassifierI*. During training, the *DecisionTreeClassifier* creates a tree where the child nodes are also instances of *DecisionTreeClassifier*. The leaf nodes contain only a single label, while the intermediate child nodes contain decision mappings for each feature. These decisions map each feature value to another *DecisionTreeClassifier*, which itself may contain decisions for another feature, or it may be a final leaf node with a classification label. The *train* class method builds this tree from the ground up, starting with the leaf nodes. It then refines itself to minimize the number of decisions needed to get to a label by putting the most informative features at the top.

To classify, the *DecisionTreeClassifier* looks at the given feature set and traces down the tree, using known feature names and values to make decisions.

Training the classifier is by a simple line of code

```
dt_classifier = DecisionTreeClassifier.train  
(training, binary=True, entropy_cutoff=0.8, depth_cutoff=5, support_cutoff=30)
```

Testing the classifier is by a simple line of code (Figure 19).

```
dt_classifier.classify(test_sentence)
```

Calculating the accuracy for the classifier is by a simple line of code (Figure 19).

```
Classifier_acc (dt_classifier, test_feats)
```

Figure 19

```
This movie is too bad to watch  
pos  
Classifier Accuracy: 0.688  
Process finished with exit code 0
```

The parameters passed in to *DecisionTreeClassifier.train* can be tweaked to improve accuracy or decrease training time. Generally, if you want to improve accuracy, you must accept a longer training time and if you want to decrease the training time, the accuracy will most likely decrease as well as we can see from Figure 19.

Functions included:

- ✚ `Label_feats_from_corpus` -> see [Commons between classifiers](#)
- ✚ `split_label_feats` -> see [Commons between classifiers](#)
- ✚ `Classifier_acc` -> see [Commons between classifiers](#)
- ✚ `Precision_recall` -> see [Commons between classifiers](#)
- ✚ `run` -> see [Commons between classifiers](#)
- ✚ `save_classifier` -> see [Commons between classifiers](#)
- ✚ `load_classifier` -> see [Commons between classifiers](#)

Code snippet:

Figure 20

```
feats = label_feat_from_corpus(movie_reviews)
training, testing = split_label_feats(feats)
# print "Running Classifier"
# run()
# print "Classifier trained"
# print "Loading Classifier"
C = load_classifier()
test_sent = "This movie is too bad to watch"
print test_sent
print C.classify(Featx.bag_of_words(test_sent))
# print C.show_most_informative_features(5)
print "Classifier Accuracy: " + str(Classifier_acc(C, testing))
dt_p, dt_r = precision_recall(C, testing)
print "Precision Positive" + str(dt_p['pos'])
print "Precision Negative" + str(dt_p['neg'])
print "Recall Positive" + str(dt_r['pos'])
print "Recall Negative" + str(dt_r['neg'])
```

Output

Figure 21

```
This movie is too bad to watch
pos
Classifier Accuracy: 0.688
Precision Positive: 0.659863945578
Precision Negative: 0.728155339806
Recall Positive: 0.776
Recall Negative: 0.6

Process finished with exit code 0
```

Phase 3c – Maxent classifier *Classifier_ME.py*

What is it?

Maximum entropy principle is as the following: Take precisely stated prior data or testable information about a probability distribution function. Consider the set of all trial probability distributions that would encode the prior data. Of those, the one with maximal information entropy is the proper distribution, according to this principle.

How it works?

Our third *classifier*, is the *maxent* or also known as *conditional exponential classifier*. It converts labeled feature sets to vectors using encoding. This encoded vector is then used to calculate *weights* for each feature that can then be combined to determine the most likely label for a feature set.

For faster training time, and more accuracy using *Python scipy algorithms* is preferable. Or we will be forced to either accept low accuracy prediction from the default algorithm, or accept much longer training time.

Like the previous classifiers, *MaxentClassifier* inherits from *ClassifierI*. Depending on the algorithm, *MaxentClassifier.train* calls one of the training functions in the *nltk.classify.maxent* module. If *scipy* is not installed, the default algorithm is *iis*, and the function used is *train_maxent_classifier_with_iis*. The other algorithm that doesn't require *scipy* is *gis*, which uses the *train_maxent_classifier_with_gis* function. **GIS** stands for **General Iterative Scaling**, while **IIS** stands for **Improved Iterative Scaling**. If *scipy* is installed, the *train_maxent_classifier_with_scipy* function is used, and the default algorithm is CG.

The basic idea behind the maximum entropy model is to build some probability distributions that fit the observed data, then choose whichever probability distribution has the highest entropy. The *gis* and *iis* algorithms do so by iteratively improving the weights used to classify features. This is where the *max_iter* and *min_lldelta* parameters come into play when training the classifier.

AGAIN ... The parameters passed in to *DecisionTreeClassifier.train* can be tweaked to improve accuracy or decrease training time. Generally, if you want to improve accuracy, you must accept a longer training time and if you want to decrease the training time, the accuracy will most likely decrease as well as we can see from Figure 22.

Training the classifier is by a simple line of code

```
me_classifier = MaxentClassifier.train  
(training, algorithm='GIS', trace=0, max_iter=1, min_lldelta=0.5)
```

Testing the classifier is by a simple line of code (Figure 22).

```
me_classifier.classify(test_sentence)
```

Calculating the accuracy for the classifier is by a simple line of code (Figure 22).

```
Classifier_acc (me_classifier, test_feats)
```

Figure 22

```
This movie is too bad to watch  
neg  
Classifier Accuracy: 0.722  
  
Process finished with exit code 0
```

Functions included:

- ✚ Label_feats_from_corpus -> see Commons between classifiers
- ✚ split_label_feats -> see Commons between classifiers
- ✚ Classifier_acc -> see Commons between classifiers
- ✚ Precision_recall -> see Commons between classifiers
- ✚ run -> see Commons between classifiers
- ✚ save_classifier -> see Commons between classifiers
- ✚ load_classifier -> see Commons between classifiers

Code snippet:

Figure 23

```
feats = label_feat_from_corps(movie_reviews)  
training, testing = split_label_feats(feats)  
# print "Running Classifier"  
# # run()  
# print "Classifier trained"  
# print "Loading Classifier"  
C = load_classifier()  
test_sent = "This movie is too bad to watch"  
print test_sent  
print C.classify(Featx.bag_of_words(test_sent))  
# # print C.show_most_informative_features(5)  
print "Classifier Accuracy: " + str(Classifier_acc(C, testing))  
me_p, me_r = precision_recall(C, testing)  
print "Precision Positive" + str(me_p['pos'])  
print "Precision Negative" + str(me_p['neg'])  
print "Recall Positive" + str(me_r['pos'])  
print "Recall Negative" + str(me_r['neg'])
```

Output

Figure 24

```
This movie is too bad to watch  
neg  
Classifier Accuracy: 0.722  
Precision Positive0.645669291339  
Precision Negative0.966386554622  
Recall Positive0.984  
Recall Negative0.46  
  
Process finished with exit code 0
```

Phase 4 – MaxVote classifier *Classifier_MV.py*

What is it?

One way to improve classification performance is to combine classifiers. The simplest way to combine multiple classifiers is to use voting, and choose whichever label gets the most votes. For this style of voting, it's best to have an odd number of classifiers so that there are no ties. This means combining at least three classifiers together. The individual classifiers should also use different algorithms; the idea is that multiple algorithms are better than one, and the combination of many can compensate for individual bias.

How it works?

The *Maximum Vote class (MaxVoteClassifier)*, is a simple class that inherits from *ClassifierI* and takes a list of **trained classifiers** you want to combine. Once created it just returns a normal working trained classifier that works the same way other classifiers do.

It's really simple:

- Test the input against all the classifiers you have.
- Take votes.
- consider taking whatever takes the most votes.
- Using odd number of classifiers are better for avoiding ties.
- Measure accuracy, precision, and recall.
- If it is not the best accuracy, it will be as good as the best of the other classifiers.

Training the classifier is by a simple line of code

```
mv_classifier = mv = MaxVoteClassifier(nb_classifier, dt_classifier, me_classifier)
```

Testing the classifier is by a simple line of code (Figure 25).

```
mv_classifier.Classifier_acc(test_sentence)
```

Calculating the accuracy for the classifier is by a simple line of code (Figure 25).

```
mv_classifier.Classifier_acc (me_classifier, test_feats)
```

Figure 25

```
This movie is too bad to watch  
neg  
Classifier Accuracy: 0.766  
  
Process finished with exit code 0
```

Functions included:

- ✚ `__init__` -> a constructor for the MaxVoteClassifier class.
- ✚ `labels` -> implementation for the interface methods
- ✚ `Classify` -> implementation for the interface methods
- ✚ `Classifier_acc` -> see Commons between classifiers
- ✚ `Precision_recall` -> see Commons between classifiers
- ✚ `save_classifier` -> see Commons between classifiers
- ✚ `load_classifier` -> see Commons between classifiers

Code snippet:

Figure 26

```
feats = Classifier_NB.label_feat_from_corps(movie_reviews)
training, testing = Classifier_NB.split_label_feats(feats)
#
nb = Classifier_NB.load_classifier()
dt = Classifier_DT.load_classifier()
me = Classifier_ME.load_classifier()
#
mv = MaxVoteClassifier(nb, dt, me)
# print mv.labels()
#
# print "Running Classifier"
# # run()
# print "Classifier trained"
print "Loading Classifier"
test_sent = "This movie is too bad to watch"
print test_sent
print mv.classify(Featx.bag_of_words(test_sent))
# # print C.show_most_informative_features(5)
print "Classifier Accuracy: " + str(mv.Classifier_acc(mv, testing))
mv_p, mv_r = Classifier_NB.precision_recall(mv, testing)
print "Precision Positive" + str(mv_p['pos'])
print "Precision Negative" + str(mv_p['neg'])
print "Recall Positive" + str(mv_r['pos'])
print "Recall Negative" + str(mv_r['neg'])
```

Output

Figure 27

```
This movie is too bad to watch
neg
Classifier Accuracy: 0.766
Precision Positive0.771428571429
Precision Negative0.760784313725
Recall Positive0.756
Recall Negative0.776

Process finished with exit code 0
```


Comparison & Conclusion:

Based on the following attributes:

- ✓ 500 test cases
- ✓ Minimum score for high information words is 5
- ✓ Decision tree *depth_cutoff* is 5
- ✓ MaxEnt Algorithm is *GIS*
- ✓ MaxEnt *maximum iteration* is 5

The following results appeared:

Classifier	Accuracy “normal”	Accuracy “after stopwords filtering”	Accuracy “testing with Bigrams”	Accuracy “with high information words”
NaiveBayes	72.8 %	72.6 %	72.6 %	91 %
DecisionTree	68.8 %	68.8 %	68.8 %	68.6 %
MaxEnt	72.2 %	72.4 %	72.4 %	91.2 %
MaxVote	76.6 %	76.6 %	72.4 %	83.4 %

Based on the following attributes (Used in the output file):

- ✓ 500 test cases
- ✓ Minimum score for high information words is 5
- ✓ Decision tree *depth_cutoff* is 10
- ✓ MaxEnt Algorithm is *GIS*
- ✓ MaxEnt *maximum iteration* is 10

The following results appeared:

Classifier	Accuracy “normal”	Accuracy “after stopwords filtering”	Accuracy “testing with Bigrams”	Accuracy “with high information words”
NaiveBayes	72.8 %	72.6 %	72.6 %	91 %
DecisionTree	68.8 %	68.8 %	68.8 %	69 %
MaxEnt	72.2 %	72.4 %	71.8 %	91.2 %
MaxVote	76.6 %	76.6 %	72.4 %	83.4 %

Based on the following attributes (Used in the output file):

- ✓ 500 test cases
- ✓ Minimum score for high information words is 10
- ✓ Decision tree *depth_cutoff* is 10
- ✓ MaxEnt Algorithm is *GIS*
- ✓ MaxEnt *maximum iteration* is 10

The following results appeared:

Classifier	Accuracy “normal”	Accuracy “after stopwords filtering”	Accuracy “testing with Bigrams”	Accuracy “with high information words”
NaiveBayes	72.8 %	72.6 %	72.6 %	88.8 %
DecisionTree	68.8 %	68.8 %	68.8 %	69.2 %
MaxEnt	72.2 %	72.4 %	72.4 %	88.4 %
MaxVote	76.6 %	76.6 %	72.4 %	81.6 %

- ❖ As we can see from the previous tables, doing tests with the most informative features has the biggest impact on the classifier’s accuracy.
- ❖ Also increasing the depth cutoff for the Decision tree module, as also improving accuracy, but as mentioned previously training time will be longer.

Test cases are too many to include all, but that was a preview for a potentially good module for text analysis, and sentiment analysis.

NOTE: if any of the training parameters changed, the classifier will need to be re-trained. So please remember to uncomment the corresponding lines of code.

***** END OF V1.0 *****

Finally:

Submitted Items for the project:

Documentation

Source Code

Corpus

Full output sample