Option 2. Processing and Classification of Sentiment or other Data
Data set: Detection of SPAM in email

Motive:

The dataset that we have was gleaned from their web site at http://www.aueb.gr/users/ion/data/enron-spam/. Corpus has two folder one with spam emails "Spam Folder" and no spam emails "Ham Folder". The motive of classification is to detect Spam emails from the Enron public email corpus.

Dataset Used:

The "raw" subdirectory contains the messages in their original form. Spam messages in non-Latin encodings, ham messages sent by the owners of the mailboxes to themselves (sender in "To:", "Cc:", or "Bcc" field), and a handful of virus-infected messages have been removed, but no other modification has been made. The messages in the "raw" subdirectory are more than the corresponding messages in the "preprocessed" subdirectory, because: (a) duplicates are preserved in the "raw" form, and (b) during the preprocessing, ham and/or spam messages were randomly subsampled to obtain the desired ham : spam ratios. See the paper for further details. http://www.aueb.gr/users/ion/docs/ceas2006_paper.pdf

Following are steps for text classification:

1. I have taken extra data sets from http://www.aueb.gr/users/ion/data/enron-spam/ to increase regular emails in the "ham" folder, and emails in the "spam" folder. I have used this because for better distribution and randomness of data which will gives us more accurate result.

2. Lower case:
   Converted text to lower case words to remove upper case and lowercase sensitivity.

   Code snapshot:
   ```
   word_list = re.split('\s+', document.lower())
   ```

3. Remove punctuation and numbers:
   Remove punctuation and numbers from text as that will deviate our result from correct analysis.

   Code snapshot:
   ```
   punctuation = re.compile(r'[-.?!,":;()|0-9]')

   word_list = [punctuation.sub("", word) for word in word_list]
   ```

4. Removed stopwords:

Create own stopword list in "stopwords_emailSpam.txt" to have more control over data. and to remove data like cc , www , %20 and many more to increase accuracy of classification. As these data are taken from "html form", it is obvious to have cc, www, % as common word in it. It is better to remove these for better analysis. A very small stop word list is probably better than a large one.

Code snapshot:

```
1.  stopwords_email = [line.strip() for line in open('stopwords_emailSpam.txt')]
2.
for word in word_list:
        if word not in stopwords_email:
            final_word_list.append(word)
```

5. **"bag-of-words"** features:

I used "bag-of-words" features to collect all the words in the corpus and select 1500 number of most frequent words to be the word features. I have changed this number to analyze classification. I will discuss more in experiment section.

Code snapshot:

```
words =  nltk.FreqDist(w.lower() for w in all_words)
word_features = words.keys()[:1500]
```

6. NLTK Naïve Bayes classifier :

I used NLTK Naïve Bayes classifier to train and test data. Initially taken 90 % of data as training set and 10% as test set.

Code snapshot:

```
training_size = int(0.1*len(featuresets))
test_set = featuresets[:training_size]
training_set = featuresets[training_size:]
classifier = nltk.NaiveBayesClassifier.train(training_set)
print "Accuracy of classifier :"
print nltk.classify.accuracy(classifier, test_set)
```

7. Most informative features:

Displayed top 50 most informative features by using "show_most_informative_features()" funtion

Code snapshot:

```
print classifier.show_most_informative_features(50)
```

8. Calculated **precision, recall and F-measure scores** for ham and spam feature. Along with this showed the confusion matrix.
Modularized the code so that same function can be called as many times.
Here, TP is true positive
FN is false negative
FP is false positive
TN is true negative
The percentage of actual yes answers that are right is called recall.
This is measured by TP / (TP + FP)
The percentage of predicted yes answers that are right is called precision.
This is measured by TP / (TP + FN)

The harmonic mean, called the F-measure is 2 * (recall * precision) / (recall + precision)

		Predicted Class	
		Class=Yes	Class=No
Actual Class	Class=Yes	a	b
	Class=No	c	d

a: TP (true positive)
b: FN (false negative)
c: FP (false positive)
d: TN (true negative)

I have computed precision and recall for both **the ham (positive) and spam (negative)** labels.

Code snapshot:

```
#########################################################################################
## Obtain precision, recall and F-measure scores. ##
#########################################################################################
def Obtain_precision_recall_and_Fmeasure_scores(classifier_type, test_set):
  reflist = []
  testlist = []
  for (features, label) in test_set:
    reflist.append(label)
    testlist.append(classifier_type.classify(features))
  print " "
  print "The confusion matrix"
  cm = nltk.metrics.ConfusionMatrix(reflist, testlist)
  print cm

#  precision and recall
# start with empty sets for true positive, true negative, false positive, false negative,

  (refpos, refneg, testpos, testneg) = (set(), set(), set(), set())

  for i, label in enumerate(reflist):
    if label == 'spam': refneg.add(i)
    if label == 'ham': refpos.add(i)
  for i, label in enumerate(testlist):
    if label == 'spam': testneg.add(i)
    if label == 'ham': testpos.add(i)

  def printmeasures(label, refset, testset):
    print label, 'precision:', nltk.metrics.precision(refset, testset)
    print label, 'recall:', nltk.metrics.recall(refset, testset)
    print label, 'F-measure:', nltk.metrics.f_measure(refset, testset)

  printmeasures('Positive_HAM ', refpos, testpos)
  print ""
  printmeasures('Negative_SPAM ', refneg, testneg)
  print ""
```

I used **cross-validation** to obtain precision, recall and F-measure scores. On every instance called Obtain_precision_recall_and_Fmeasure_scores() to display precision, recall and F-measure scores. This method is called *cross-validation*, or sometimes *k-fold cross-validation*. In this method, We first randomly partition the development data into k subsets, each approximately equal in size. Then we train the classifier k times, where at each iteration, we use each subset in turn as the test set and the others as a training set. I choose a number of folds, 5, 8 and 10 folds for my experiment.

Code snapshot:

```
#########################################################################################
## cross-validation ##
#########################################################################################
# this function takes the number of folds, the feature sets
# it iterates over the folds, using different sections for training and testing in turn
#   it prints the accuracy for each fold and the average accuracy at the end
def cross_validation(num_folds, featuresets):
    subset_size = len(featuresets)/num_folds
    accuracy_list = []
    print "Running cross_validation for classifier :"
    # iterate over the folds
    for i in range(num_folds):
        print "--------------------------------------------------"
        test_this_round = featuresets[i*subset_size:][:subset_size]
        train_this_round = featuresets[:i*subset_size] + featuresets[(i+1)*subset_size:]
        # train using train_this_round
        classifier = nltk.NaiveBayesClassifier.train(train_this_round)
        # evaluate against test_this_round and save accuracy
        accuracy_this_round = nltk.classify.accuracy(classifier, test_this_round)
        print 'Accuracy Round ', i, ' = ' ,  accuracy_this_round
        Obtain_precision_recall_and_Fmeasure_scores(classifier,test_this_round)
        accuracy_list.append(accuracy_this_round)
    # find mean accuracy over all rounds
    print 'Mean accuracy over all rounds = ', sum(accuracy_list) / num_folds
```

10. using Bigram features along with unigram features

I have worked on generating bigram feature from documents. To get high frequent bigrams, I have **filter our special characters** as well as **filter by frequency**. I have got top 2000 **bigram pmi measure** and **chi-squared measure** and then sample randomly to get 2000 bigram word features for feature extraction process. I have used the nbest function which just returns the highest scoring bigrams, using the number specified in both the measures.

Code:

```
def get_bigram_word_features(hamtexts , spamtexts):
  print "-----------------------------------------------------"
  print "Getting all words and create word features"
  # create the bigram finder on the movie review words in sequence
  words = ""
  for spam in hamtexts:
    spam1 = Pre_processing_documents(spam)
    words += spam1

  for spam in spamtexts:
    spam1 = Pre_processing_documents(spam)
    words += spam1

  bigram_measures = nltk.collocations.BigramAssocMeasures()
  finder = BigramCollocationFinder.from_words(words.split(),window_size = 4)
  print finder
  finder.apply_freq_filter(6)
  top20 = finder.nbest(bigram_measures.pmi,3000)
  bigram_features = finder.nbest(bigram_measures.chi_sq, 3000)
  print " Applying bigram_measures.pmi"
  #print top20[:20]
  print " Applying bigram_measures.chi_sq"
  #print bigram_features[:20]
  return bigram_features[:2000]

def bigram_document_features(document,c,unigram_word_feature,bigram_word_feature):
  document_words = set(document)
  document_bigrams = nltk.bigrams(document)
  features = {}
  for word in unigram_word_feature:
    features['contains(%s)' % word] = (word in document_words)
  for bigram in bigram_word_feature:
    features['bigram(%s %s)' % bigram] = (bigram in document_bigrams)
  return features
```

11. tfidf scores as the values of the word features, instead of Boolean values

frequency–inverse document frequency(tf–idf) is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. It is often used as a weighting factor in information retrieval and text mining. The tf-idf value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general.

Variations of the tf–idf weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query. tf–idf can be successfully used for stop-words filtering in various subject fields including text summarization and classification.

I have created method to calculate tfidf score. Applied these to both unigram and bigram feature sets to extract information.

Code:

References: https://en.wikipedia.org/wiki/Tf%E2%80%93idf

```
##########################################################################################
# Calculating  tfidf scores
##########################################################################################
def freq(word, doc):
    return doc.count(word)


def word_count(doc):
    return len(doc)


def tf(word, doc):
    return (freq(word, doc) / float(word_count(doc)))


def num_docs_containing(word, list_of_docs):
    count = 0
    for document in list_of_docs:
        if freq(word, document) > 0:
            count += 1
    return 1 + count


def idf(word, list_of_docs):
    return math.log(len(list_of_docs) /
            float(num_docs_containing(word, list_of_docs)))

def tf_idf(word, doc, list_of_docs):
    return (tf(word, doc) * idf(word, list_of_docs))
```

Code:

```
##########################################################################################
# use word frequency or tfidf scores as the values of the word features, instead of Boolean values
##########################################################################################

tfidf_csv_file = open('tfidfWordFeatures.csv', 'wb')
tfidf_writer = csv.writer(tfidf_csv_file, quoting = csv.QUOTE_ALL)

def get_tfidf_word_features_sets(email, category,bigram_word_feature,unigram_word_feature,emaildocs):
    email_words = set(email);
    document_bigrams = nltk.bigrams(email)
    features = {}
    email_csv = []
    weakPos = 0
    for word in bigram_word_feature:
        score = tf_idf(word, email, emaildocs)
        features['bigram(%s %s)' % word] = score
        if word == 'Category':
            email_csv.append(category)
        elif word in email_words:
            email_csv.append("true")
        else:
            email_csv.append("false")

    for word in unigram_word_feature:
        score = tf_idf(word, email, emaildocs)
        features['unigram(%s)'% word] = score
        if word == 'Category':
            email_csv.append(category)
        elif word in email_words:
            email_csv.append("true")
        else:
            email_csv.append("false")
    tfidf_writer.writerow(email_csv)
    return features
```

12. csv file for testing in weka:

I have created two csv file for testing in weka one by other complete corpus data that I have gathered from web http://www.aueb.gr/users/ion/data/enron-spam/. And one that already present as corpus for this experiment.

File name:
 unigramWordFeatures.csv
 unigramWordFeatures_compelete.csv

I have done following experiment on these cvs file.

1. I have used "unigramWordFeatures.csv" file in Weka and applied Naïve Bayes classifier and use percentage split of 80%, 90% and 50% to get the result.
2. I have used "unigramWordFeatures.csv" file in Weka and applied Naïve Bayes classifier and use percentage split of 80% and cross validation fold 8 to get the result.
3. I have used "unigramWordFeatures.csv" as training set and "unigramWordFeatures_compelete.csv" as test set to compare result.
4. I have checked F-measures on two trees classifier Decision Stump and j48
5. I have checked F-measures on three functions classifier multilayer perceptron, voted perceptron and simple logistic.
6. I have checked F-measures on one rules classifier Decision table

Experiments:

1. **Comparison before and after applying Filter by stopwords or other pre-processing methods**

	Bag of words	Bigram plus unigram	tfidf
Before filter and pre-processing	0.845 accuracy	0.87 accuracy	0.875 accuracy
After filter and pre-processing	0.92 accuracy	0.96 accuracy	0.94 accuracy

Observation:
Applying pre-processing and stopwords improved accuracy in all classifier. This is because pre-processing and stopwords filter removed unnecessary words from classifier.

2. **Comparison on different sizes of vocabularies**

sizes of vocabularies	Bag of words	Bigram plus unigram	tfidf
100	0.78 accuracy	0.81 accuracy	0.84 accuracy
1000	0.84 accuracy	0.80 accuracy	0.92 accuracy
2000	0.901 accuracy	0.92 accuracy	0.97 accuracy

Observation:
Increase in size of vocabularies show increase in accuracy. This is because of bigger feature sets to classify data.

3. **Comparison using cross-validation**

Changing cross-validation number of folds	Bag of words	Bigram plus unigram	tfidf
10	0.910 avg accuracy	0.937 avg accuracy	0.93 avg accuracy
8	0.902 avg accuracy	0.935 avg accuracy	0.926 avg accuracy
5	0.90 avg accuracy	0.932 avg accuracy	0.92 avg accuracy

Observation:
Increase in number of folds increases in mean accuracy in all classifier. This is because of better distribution of calculating feature set.

Displaying 10 fold accuracy result:

cross-validation accuracy using 10 as number of folds	Bag of words	Bigram plus unigram	tfidf
1	0.905 avg accuracy	0.92 avg accuracy	0.93 avg accuracy
2	0.92 avg accuracy	0.901 avg accuracy	0.94 avg accuracy

3	0.92 avg accuracy	0.93 avg accuracy	0.92 avg accuracy
4	0.89 avg accuracy	0.92 avg accuracy	0.93 avg accuracy
5	0.90 avg accuracy	0.94 avg accuracy	0.93 avg accuracy
6	0.89 avg accuracy	0.93 avg accuracy	0.92 avg accuracy
7	0.90 avg accuracy	0.92 avg accuracy	0.93 avg accuracy
8	0.91 avg accuracy	0.91 avg accuracy	0.92 avg accuracy
9	0.92 avg accuracy	0.92 avg accuracy	0.92 avg accuracy
10	0.91 avg accuracy	0.94 avg accuracy	0.92 avg accuracy

4. Measure accuracy , precision , recall and f- measures score in all three classifier

	Bag of words	Bigram plus unigram	tfidf
Accuracy	0.921	0.93	0.955
Precision score	0.937	0.942	0.97
Recall score	0.89	0.90	0.92
F-measure score	0.924	0.934	0.955

Observation:

Bigram plus unigram is performance better in classification than bag of words because of extra feature add to classification F-measure and accuracy better in tfidf than Bigram plus unigram because it gives lesser value to must frequent words which almost eliminate most filtering processes. Plus, we observed recall score lesser than F-measure which is less than Precision.

5. Using weka

	Precision	Recall	F-measures
Naïve Bayes classifier and use percentage split of 50%	0.922	0.905	0.904
Naïve Bayes classifier and use percentage split of 80%	0.955	0.95	0.95
Naïve Bayes classifier and use percentage split of 90%	0.9	0.919	0.95
Naïve Bayes classifier and use percentage split of 80% and cross validation fold 8	0.955	0.95	0.95
used “unigramWordFeatures.csv” as training set and “unigramWordFeatures_compelete.csv” as test set using Naïve Bayes	0.861	0.858	0.857
Trees classifier Decision Stump	0.803	0.723	0.703
Trees classifier j48	0.911	0.91	0.91
Functions classifier multilayer perceptron	0.944	0.943	0.942

Functions classifier voted perceptron	0.915	0.91	0.91
Functions classifier simple logistic	0.95	0.948	0.947
Rules classifier Decision table	0.904	0.895	0.894

Observation:

1. We observed that Decision Stump classifier performance poor of all because it uses only “Thanks” to classify the text.

2. Out of mentioned tree classifier, j48 tree classifier performance better than Decision Stump.

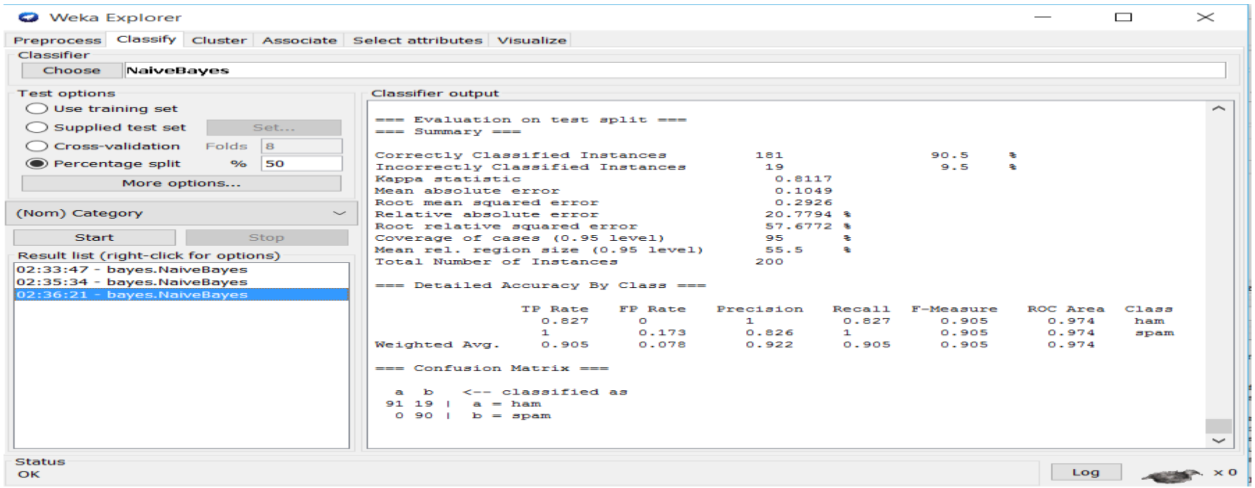
3. All Function classifier have almost near values of F-measures. Among this simple logistic perceptron performed better of all.

4. Function classifier performed better than Rules classifier.

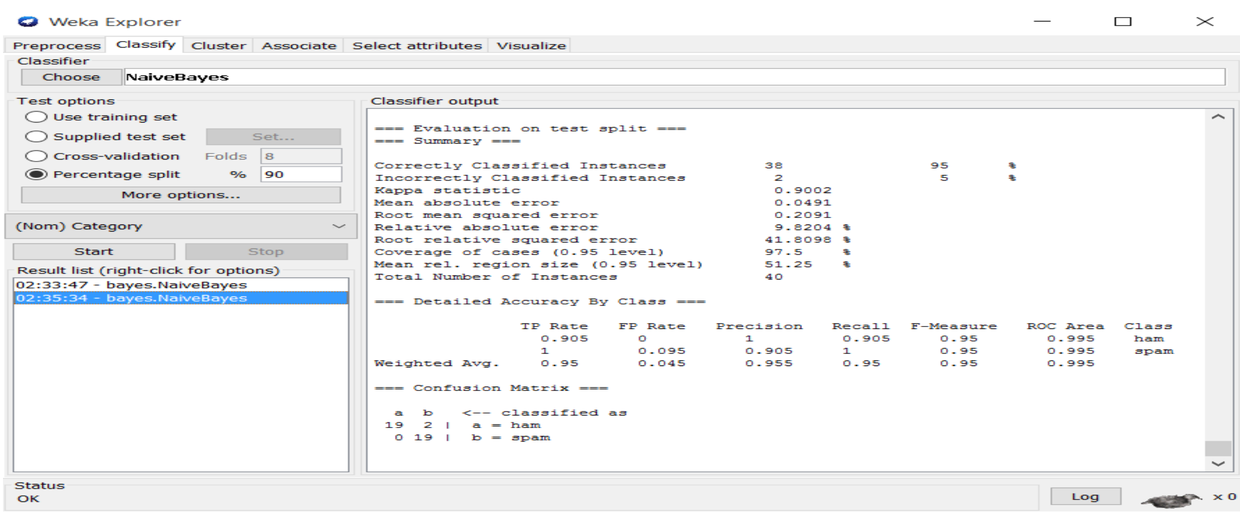
5. In using “unigramWordFeatures.csv” as training set and “unigramWordFeatures_compelete.csv” as test set using Naïve Bayes, we observed F-measures decreases to 0.857 from 0.95. In splitting of same datasets into training and test set with split 80% and cross fold as 8 we got 0.95 as F-measures. This may be due to extra classifier word has not be taken into account from larger data sets (complete corpus folder).

Sample output Weka:

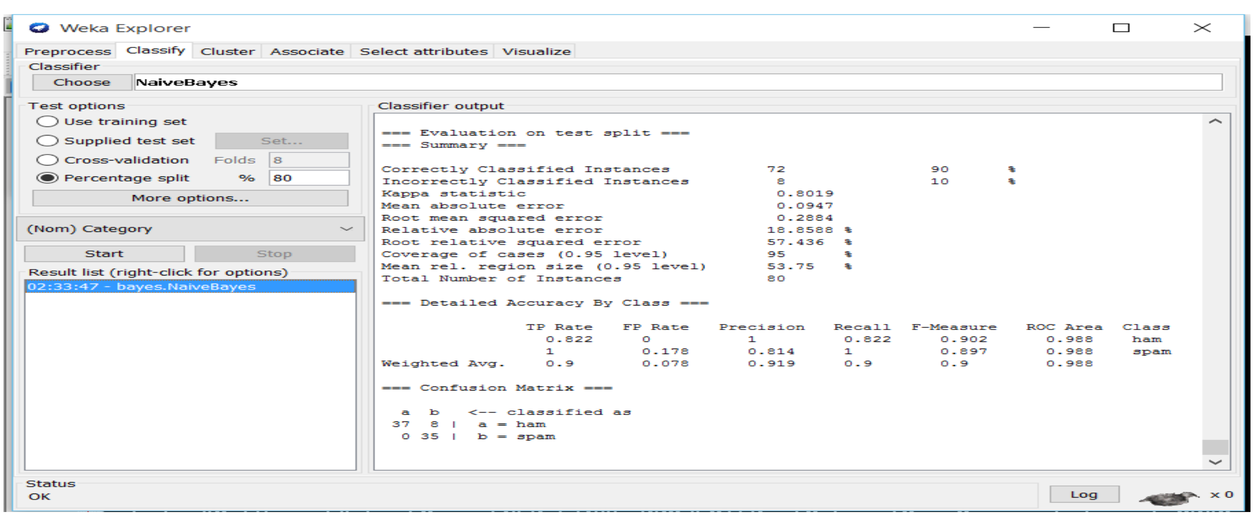
1. Naïve Bayes classifier and use percentage split of 50%



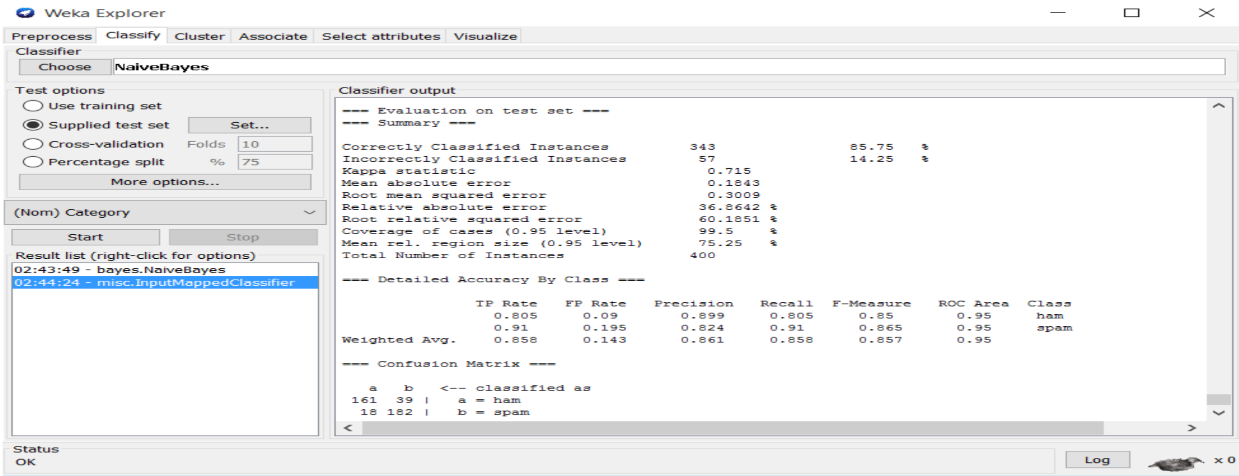
2. Naïve Bayes classifier and use percentage split of 90%



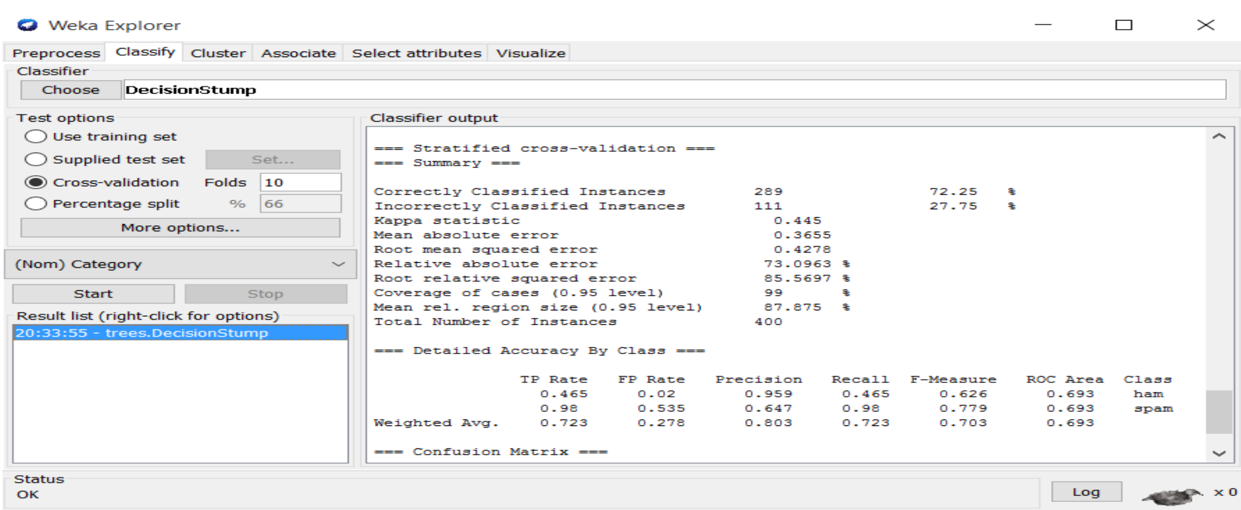
3. Naïve Bayes classifier and use percentage split of 80%



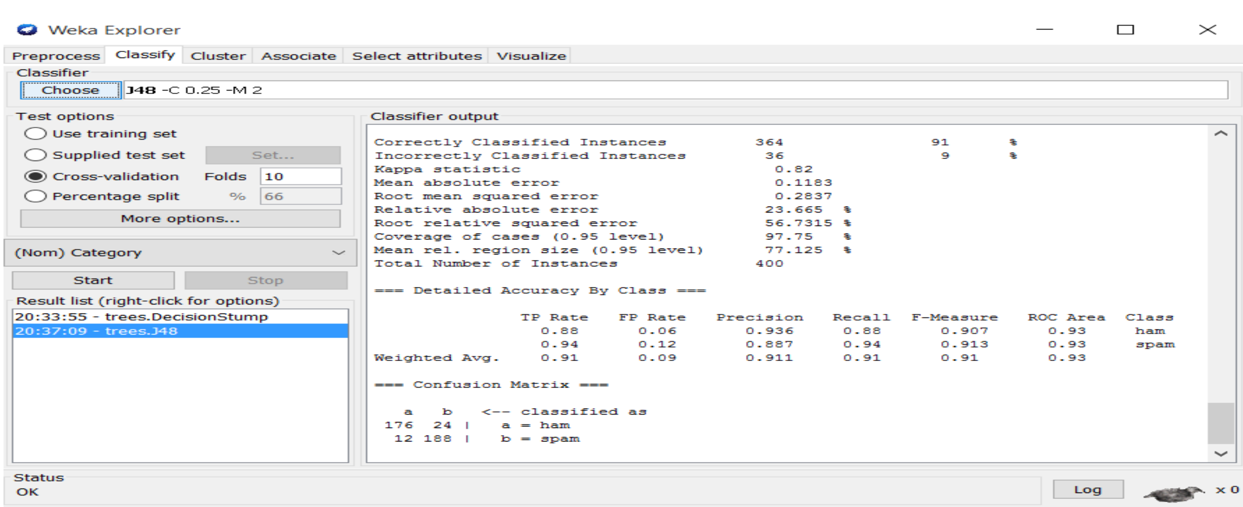
4. used “unigramWordFeatures.csv” as training set and “unigramWordFeatures_compelete.csv” as test set by using Naïve Bayes



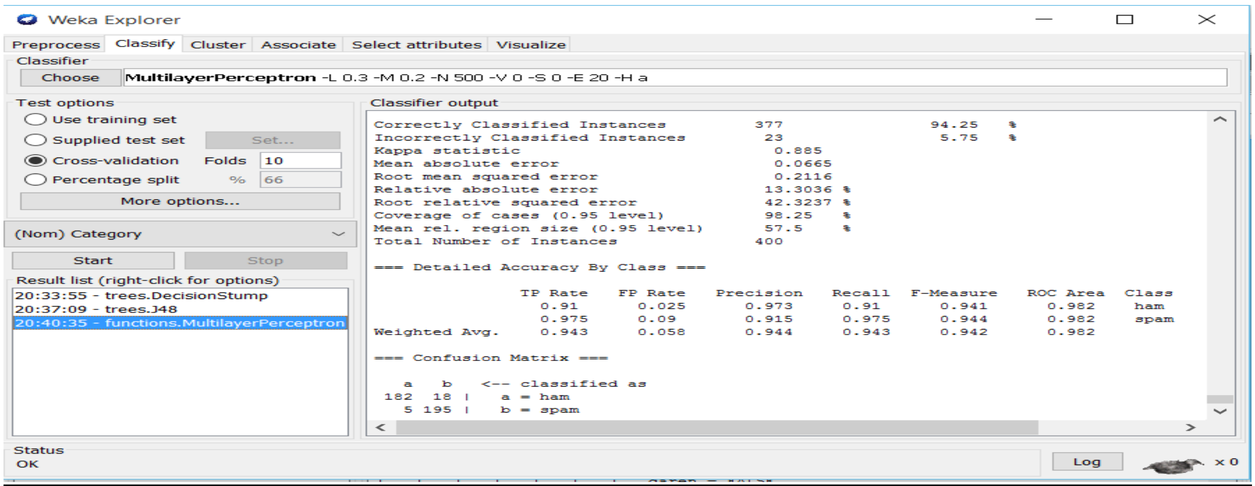
5. Trees classifier Decision Stump



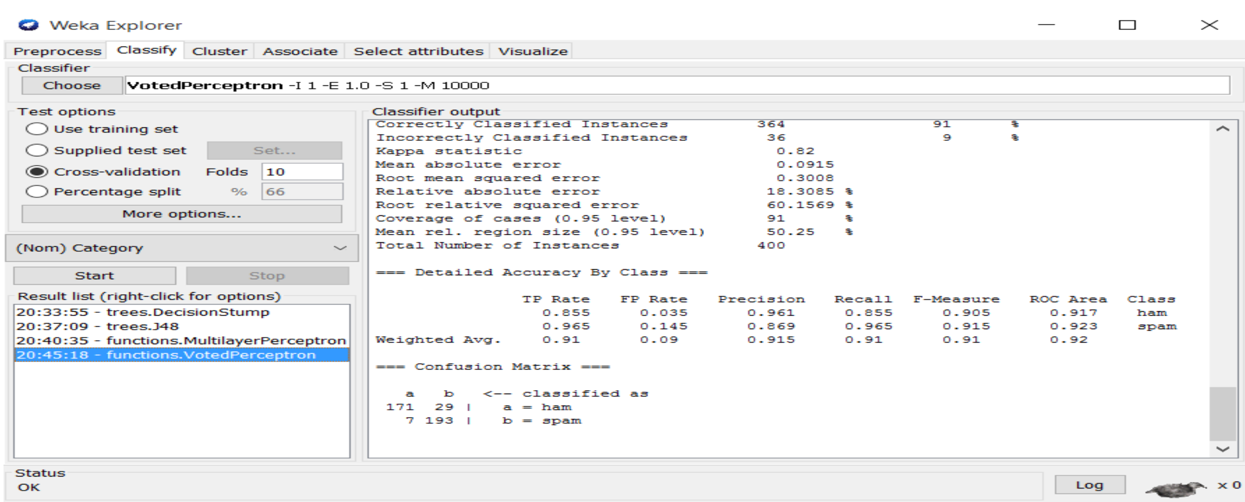
6. Trees classifier j48



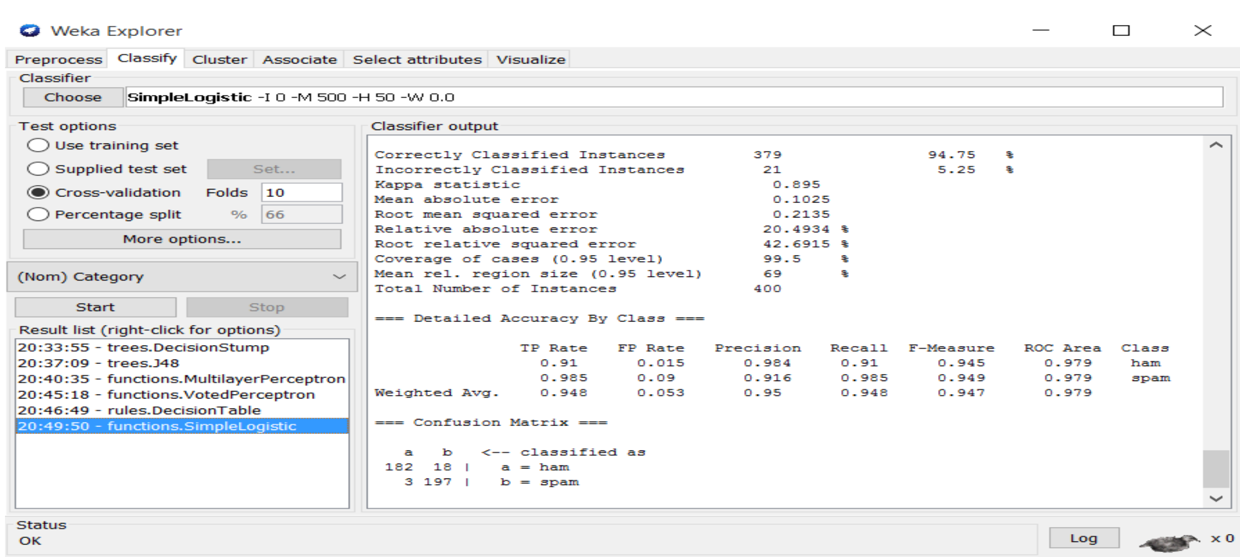
7. Functions classifier multilayer perceptron



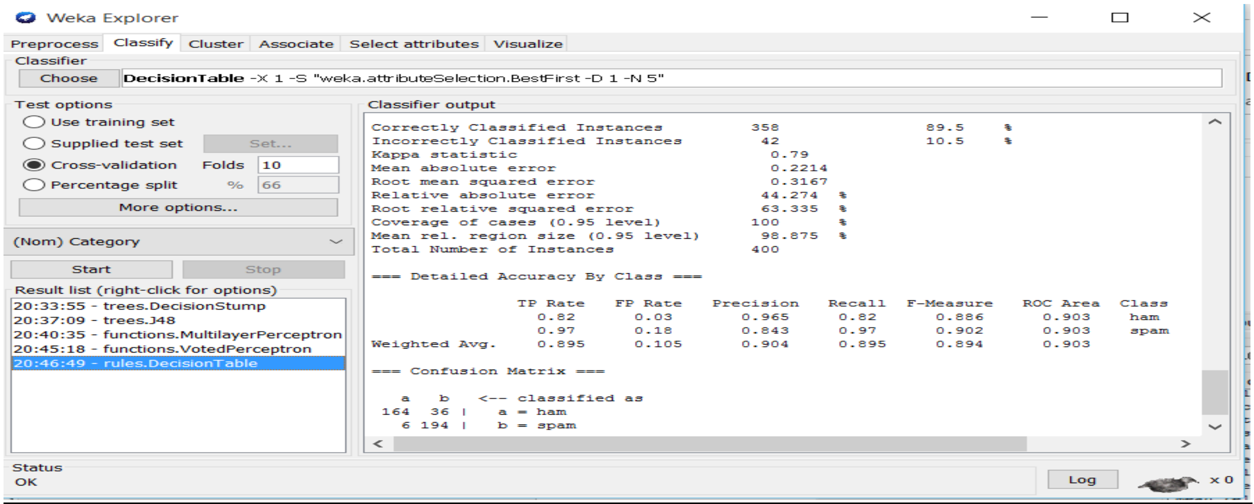
8. Functions classifier voted perceptron



9. Functions classifier simple logistic



10. Rules classifier Decision table



References:

1. https://en.wikipedia.org/wiki/Tf%E2%80%93idf
2. http://www.aueb.gr/users/ion/docs/ceas2006_paper.pdf
3. http://www.aueb.gr/users/ion/data/enron-spam/