**IST_664 NLP_FINAL_PROJECT**

**Detection of SPAM in email** **UdayVempalli (696526545)**

## 1.INTRODUCTION:

The dataset used for the project is **Detecting spam emails** from the Enron public email corpus. The dataset was created by augmenting the existing Enron corpus with additional spam emails to ensure an adequate number of spam examples for training a classifier. The non-spam emails in the dataset are labeled as "ham". The information about the dataset's source and details can be found in a research paper and on the website of the Athens University of Economics and Business (AUEB).

The dataset was obtained from the AUEB website at http://www.aueb.gr/users/ion/data/enron-spam/. Although there are three large directories containing both spam and ham emails, this particular analysis focuses on the first directory. The "ham" folder within the first directory contains 3,672 regular, non-spam emails, while the "spam" folder contains 1,500 emails that are classified as spam.

The Enron public email corpus is a widely used dataset for email analysis and research. It was derived from the internal email communication of the Enron Corporation, a major energy company that filed for bankruptcy in 2001. The dataset is valuable for studying various aspects of email data, including spam detection.

To enhance the dataset for spam detection purposes, additional spam emails were introduced into each user's email stream. This augmentation aimed to increase the number of spam examples available for training a spam classifier. By combining the original Enron corpus with these injected spam emails, the dataset provides a diverse collection of both ham and spam emails for analysis and classification.

The dataset is structured into different directories, with each directory containing a set of emails. In this case, the analysis focuses on the first directory, which is comprised of two subdirectories: "ham" and "spam". The "ham" subdirectory consists of 3,672 regular, non-spam emails, while the "spam" subdirectory contains 1,500 emails classified as spam.

The dataset's availability on the AUEB website allows researchers and practitioners to access and utilize it for various purposes, including training and evaluating spam detection algorithms and developing email-related NLP (Natural Language Processing) models. The provided link to the research paper (http://www.aueb.gr/users/ion/docs/ceas2006_paper.pdf) offers further details about the dataset and its creation process, providing additional insights for anyone interested in exploring or working with the Enron spam detection dataset.

Overall, the dataset described in the paragraph provides a subset of the Enron public email corpus, specifically curated for spam detection purposes. With a significant number of ham and spam emails, it offers a valuable resource for developing and evaluating spam detection models, contributing to advancements in email analysis and filtering techniques.

## 2. Text Classification Tasks:

### I. Processing the text and Tokenization:

```
# open python and nltk packages needed for processing
import os
import sys
import random
import nltk
from nltk.corpus import stopwords
```

These imports enable various functionalities for preprocessing and analyzing text data in Python using the NLTK package.

```
# function to read spam and ham files, train and test a classifier
def processspamham(dirPath,limitStr):
  # convert the limit argument from a string to an int
  limit = int(limitStr)

  # start lists for spam and ham email texts
  hamtexts = []
  spamtexts = []
  os.chdir(dirPath)
  # process all files in directory that end in .txt up to the limit
  #    assuming that the emails are sufficiently randomized
  for file in os.listdir("./spam"):
    if (file.endswith(".txt")) and (len(spamtexts) < limit):
      # open file for reading and read entire file into a string
      f = open("./spam/"+file, 'r', encoding="latin-1")
      spamtexts.append (f.read())
      f.close()
  for file in os.listdir("./ham"):
    if (file.endswith(".txt")) and (len(hamtexts) < limit):
      # open file for reading and read entire file into a string
      f = open("./ham/"+file, 'r', encoding="latin-1")
      hamtexts.append (f.read())
      f.close()

  # print number emails read
  #print ("Number of spam files:",len(spamtexts))
 # print ("Number of ham files:",len(hamtexts))


  # create list of mixed spam and ham email documents as (list of words, label)
  emaildocs = []
  # add all the spam
  for spam in spamtexts:
    tokens = nltk.word_tokenize(spam)
    emaildocs.append((tokens, 'spam'))
  # add all the regular emails
  for ham in hamtexts:
    tokens = nltk.word_tokenize(ham)
    emaildocs.append((tokens, 'ham'))

  # randomize the list
  random.shuffle(emaildocs)
  return emaildocs
  # print a few token lists
  #for email in emaildocs[:1]:
   # print (email)
```

The above code defines a function **processspamham** that processes spam and ham files from a specified directory. Here's a brief explanation of the code:

1. The function takes two arguments: **dirPath**, which is the directory path where the spam and ham files are located, and **limitStr**, which is a string specifying the limit on the number of files to be processed.
2. The **limitStr** is converted to an integer **limit** using the **int()** function.
3. Empty lists **hamtexts** and **spamtexts** are initialized to store the text contents of ham and spam emails, respectively.
4. The current working directory is changed to the specified **dirPath** using **os.chdir()**.
5. The code loops through all the files in the "spam" directory, reads the contents of each file, and appends the text to the **spamtexts** list. The process is repeated for the "ham" directory and the **hamtexts** list.
6. Next, a list called **emaildocs** is created to store the mixed spam and ham email documents as tuples of the form **(list of words, label)**. Each email text is tokenized using **nltk.word_tokenize()** to split it into individual words, and the tokenized text along with its label ('spam' or 'ham') is appended to the **emaildocs** list.
7. The **emaildocs** list is then shuffled randomly using **random.shuffle()** to mix the order of the email documents.
8. Finally, the **emaildocs** list is returned as the output of the **processspamham** function.

This code essentially reads the spam and ham files from the specified directory, tokenizes the email texts, assigns the appropriate labels, and randomizes the order of the email documents. The processed data in the form of tokenized email texts and their respective labels can be further used for training and testing a classifier to detect spam emails.

```
direc = 'C:\\Users\\udayv\\Downloads\\FinalProject_Data_N\\FinalProjectData\\EmailSpamCorpora\\corpus'
#passing the directory and the no of files to be picked up from it.
res = processspamham(direc, 1000)
```

the function **processspamham()** will process the spam and ham files in the specified directory path, limiting the number of files to be processed to 1000. The resulting processed email documents will be stored in the variable **res** for further analysis or modeling.

```
all_words_list = [word for (sent,cat) in res for word in res]
```

```
new_list = []
for x in all_words_list:
    for i in x:
        new_list.append(i)
```

```
result = []
for i in new_list:
    for j in i:
        if len(j) > 2 and j not in ('ham','spam','Subject'):
            result.append(j)
```

The result of this code is a flat list **all_words_list** that contains all the words extracted from the sentence-category tuples in the **res** list. This construct allows you to gather all the words from multiple sentences into a single list, which can be useful for various text analysis tasks such as feature extraction or building a vocabulary for a language model.

```
all_words = nltk.FreqDist(result)
all_words
```

```
FreqDist({'the': 18816000, 'ect': 10980000, 'and': 10032000, 'for': 7700000, 'you': 6926000, 'this': 5712000, 'hou': 569800
0, 'your': 3802000, 'that': 3620000, '2000': 3410000, ...})
```

The provided code calculates the frequency distribution of words in a list called **result** using the **nltk.FreqDist()** function.

The resulting **all_words** object can be used to retrieve information about the frequency of words in the **result** list. It allows you to access the count of individual words, as well as perform various operations such as sorting the words by frequency or generating a plot of the frequency distribution.

```
# get the 3000 most frequently appearing keywords in the corpus
word_items = all_words.most_common(3000)
word_features = [word for (word,count) in word_items]
print(word_features[:100])
```

```
['the', 'ect', 'and', 'for', 'you', 'this', 'hou', 'your', 'that', '2000', 'enron', 'with', 'will', 'have', 'from', 'are',
'please', 'not', 'com', 'all', 'our', 'subject', 'meter', 'can', 'gas', 'deal', 'any', 'http', '000', 'has', 'corp', 'new',
'thanks', 'get', 'was', 'know', 'here', 'need', 'more', 'forwarded', 'out', 'only', 'may', 'daren', 'hpl', 'there', 'informa
tion', 'these', 'into', 'company', 'mmbtu', 'www', 'let', 'time', 'would', 'but', 'been', 'price', 'one', 'should', 'now',
'month', 'mail', 'contract', 'email', 'nbsp', 'sitara', 'what', 'see', 'which', 'day', 'also', 'volume', 'they', 'about', 'f
ree', 'like', 'energy', 'deals', 'font', 'their', 'pills', 'some', 'change', 'ami', 'business', 'just', 'message', 'want',
'volumes', '2004', 'over', 'attached', 'other', 'farmer', 'xls', 'its', 'questions', 'contact', 'who']
```

The above code retrieves the 3000 most frequently appearing keywords from the **all_words** frequency distribution and stores them in the list **word_features**. Here's a breakdown of the code:

By executing this code, we will obtain the 3000 most frequently appearing keywords in the corpus. These keywords are stored in the **word_features** list, and the first 100 words from the list will be printed to the console. These keywords can be used as features for various text classification.

```
# define features (keywords) of a document for a BOW/unigram baseline
# each feature is 'contains(keyword)' and is true or false depending
# on whether that keyword is in the document
def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['V_{}'.format(word)] = (word in document_words)
    return features
```

The above code defines a function **document_features** that generates features (keywords) for a document based on a Bag-of-Words (BOW) or unigram baseline approach.Here's an explanation of the code:

1. **document_features(document, word_features)**: This function takes two arguments: **document**, which represents a list of words in a document, and **word_features**, which is a list of keywords or features.
2. **document_words = set(document)**: Creates a set **document_words** to store the unique words in the document. This allows for efficient membership testing.
3. **features = {}**: Initializes an empty dictionary **features** to store the features of the document.
4. **for word in word_features:**: Iterates over each word in the **word_features** list.
5. **features['V_{}'.format(word)] = (word in document_words)**: Creates a feature for the document using the word as the feature name. The feature is represented by a key-value pair in the **features** dictionary. The key is constructed as **'V_'** followed by the word, and the value is a boolean indicating whether the word is present in the document (**True**) or not (**False**).
6. **return features**: Returns the dictionary **features** containing the features of the document.

By using this function, we can generate a feature set for a document by checking whether each keyword in the **word_features** list is present in the document or not. The resulting feature set is represented as a dictionary, where the keys are the feature names and the values indicate the presence or absence of each feature in the document. This type of feature representation can be

useful for training machine learning models or conducting text classification tasks based on the Bag-of-Words approach.

```
# training using naive Baysian classifier, training set is approximately 90% of data
train_set, test_set = featuresets[100:], featuresets[:100]
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

```
# evaluate the accuracy of the classifier
nltk.classify.accuracy(classifier, test_set)
```

```
17]: 0.95
```

The abovecode splits the **featuresets** into training and test sets, trains a Naive Bayes classifier on the training set, and evaluates the accuracy of the classifier on the test set. Here's an explanation of the code:

1. **train_set, test_set = featuresets[100:], featuresets[:100]**: Splits the **featuresets** into a training set and a test set. The training set (**train_set**) contains all the feature sets from index 100 onwards, and the test set (**test_set**) contains the first 100 feature sets.
2. **classifier = nltk.NaiveBayesClassifier.train(train_set)**: Trains a Naive Bayes classifier using the **nltk.NaiveBayesClassifier** class. The **train()** method is called on the classifier object, and it takes the **train_set** as input. This step trains the classifier using the labeled feature sets in the training set.
3. **nltk.classify.accuracy(classifier, test_set)**: Evaluates the accuracy of the trained classifier on the test set using the **nltk.classify.accuracy()** function. This function takes two arguments: the trained **classifier** and the **test_set**. It returns the accuracy of the classifier on the test set as a decimal value between 0 and 1.

By executing this code, we will obtain the accuracy of the Naive Bayes classifier on the test set. This accuracy represents how well the classifier performs in predicting the labels (e.g., 'spam' or 'ham') of the test instances based on the features extracted from the training instances.

## II. Filtering By Stop Words

## Filtering by Stop Words

```
stopwords = nltk.corpus.stopwords.words('english')
len(stopwords)
```

```
179
```

```
# this list of additional stop words includes some other words like hardly and rarely
additionalstopwords = ['no', 'not', 'never', 'none', 'nowhere', 'nothing', 'noone', 'rather', 'hardly',
                       'scarcely', 'rarely', 'seldom', 'neither', 'nor']
```

```
newstopwords = [word for word in stopwords if word not in additionalstopwords]
len(newstopwords)
#newstopwords
```

```
176
```

```
len(result)
```

```
523866000
```

```
new_all_words_list = [word for word in result if word not in newstopwords]
len(new_all_words_list)
```

```
414800000
```

```
new_all_words = nltk.FreqDist(new_all_words_list)
len(new_all_words)
```

```
36469
```

```
new_word_items = new_all_words.most_common(2000)
```

```
new_word_features = [word for (word,count) in new_word_items]
```

```
print(new_word_features[:50])
```

```
['ect', 'hou', '2000', 'enron', 'please', 'not', 'com', 'subject', 'meter', 'gas', 'deal', 'http', '000', 'corp', 'new', 'th
anks', 'get', 'know', 'need', 'forwarded', 'may', 'daren', 'hpl', 'information', 'company', 'mmbtu', 'www', 'let', 'time',
'would', 'price', 'one', 'month', 'mail', 'contract', 'email', 'nbsp', 'sitara', 'see', 'day', 'also', 'volume', 'free', 'li
ke', 'energy', 'deals', 'font', 'pills', 'change', 'ami']
```

```
# get features sets for a document, including keyword features and category feature
featuresetsWithStop = [(document_features(d, new_word_features), c) for (d, c) in res]
```

```
# get features sets for a document, including keyword features and category feature
featuresetsWithStop = [(document_features(d, new_word_features), c) for (d, c) in res]
```

```
# training using naive Baysian classifier, training set is approximately 90% of data
train_set, test_set = featuresetsWithStop[100:], featuresetsWithStop[:100]
classifier    = nltk.NaiveBayesClassifier.train(train_set)
```

```
# evaluate the accuracy of the classifier
nltk.classify.accuracy(classifier, test_set)
```

```
0.95
```

The provided code initializes a list of stopwords, adds additional stopwords, creates a new list of stopwords excluding the additional ones, filters the original word list by removing the stopwords, calculates the frequency distribution of the filtered words,

selects the most common 2000 words as features, and finally creates a new set of featuresets using the filtered word features.

By executing this code, you will obtain a filtered set of word features without the stopwords, and a new set of featuresets based on the filtered word features. These updated featuresets can be used for training and evaluating a classifier to potentially improve the performance by excluding common stopwords from the analysis.

III. **Experimenting with BiGraphs:**

```
####   adding Bigram features   ####
from nltk.collocations import *
bigram_measures = nltk.collocations.BigramAssocMeasures()
```

```
finder = BigramCollocationFinder.from_words(result)
```

```
bigram_features = finder.nbest(bigram_measures.chi_sq, 500)
```

The above code imports the **nltk.collocations** module and specifically imports the **BigramAssocMeasures** class from it. This class is used to calculate various association measures for bigram collocations (pairs of adjacent words) in a text corpus.

By using the **BigramAssocMeasures** class, you can calculate association measures such as pointwise mutual information (PMI) and chi-square for bigram collocations in our text corpus.

```
finder = BigramCollocationFinder.from_words(result)
```

```
bigram_features = finder.nbest(bigram_measures.chi_sq, 500)
```

```
# define features that include words as before
# add the most frequent significant bigrams
# this function takes the list of words in a document as an argument and returns a feature dictionary
# it depends on the variables word_features and bigram_features
def bigram_document_features(document, word_features, bigram_features):
    document_words = set(document)
    document_bigrams = nltk.bigrams(document)
    features = {}
    for word in word_features:
        features['V_{}'.format(word)] = (word in document_words)
    for bigram in bigram_features:
        features['B_{}_{}'.format(bigram[0], bigram[1])] = (bigram in document_bigrams)
    return features
```

By using this **bigram_document_features** function, you can generate a feature set for a document by considering both unigram and bigram features. The resulting feature set

includes features that indicate the presence or absence of individual words (unigrams) and significant word pairs (bigrams) in the document.

```
# features in document 0
print(bigram_featuresets[0][0])
```

```
{'V_the': True, 'V_ect': False, 'V_and': False, 'V_for': True, 'V_you': False, 'V_this': False, 'V_hou': False, 'V_your':
False, 'V_that': False, 'V_2000': False, 'V_enron': False, 'V_with': False, 'V_will': False, 'V_have': False, 'V_from': F
alse, 'V_are': False, 'V_please': False, 'V_not': False, 'V_com': False, 'V_all': False, 'V_our': False, 'V_subject': Fal
se, 'V_meter': False, 'V_can': False, 'V_gas': False, 'V_deal': False, 'V_any': False, 'V_http': False, 'V_000': False,
'V_has': False, 'V_corp': False, 'V_new': False, 'V_thanks': False, 'V_get': False, 'V_was': False, 'V_know': False, 'V_h
ere': False, 'V_need': False, 'V_more': False, 'V_forwarded': False, 'V_out': False, 'V_only': False, 'V_may': False, 'V_
daren': False, 'V_hpl': False, 'V_there': False, 'V_information': False, 'V_these': False, 'V_into': False, 'V_company':
False, 'V_mmbtu': False, 'V_www': False, 'V_let': False, 'V_time': False, 'V_would': False, 'V_but': False, 'V_been': Fal
se, 'V_price': False, 'V_one': False, 'V_should': False, 'V_now': False, 'V_month': False, 'V_mail': False, 'V_contract':
False, 'V_email': False, 'V_nbsp': False, 'V_sitara': True, 'V_what': False, 'V_see': False, 'V_which': False, 'V_day': F
alse, 'V_also': False, 'V_volume': True, 'V_they': False, 'V_about': False, 'V_free': False, 'V_like': False, 'V_energy':
False, 'V_deals': False, 'V_font': False, 'V_their': False, 'V_pills': False, 'V_some': False, 'V_change': False, 'V_am
i': False, 'V_business': False, 'V_just': False, 'V_message': False, 'V_want': False, 'V_volumes': False, 'V_2004': Fals
e, 'V_over': False, 'V_attached': False, 'V_other': False, 'V_farmer': False, 'V_xls': False, 'V_its': False, 'V_question
s': False, 'V_contact': False, 'V_who': False, 'V_when': False, 'V_make': False, 'V_width': False, 'V_report': False, 'V_
call': False, 'V_list': False, 'V_click': False, 'V_back': False, 'V_money': False, 'V_100': False, 'V_texas': False, 'V_
could': False, 'V_nom': False, 'V_set': False, 'V_chokshi': False, 'V_inc': False, 'V_robert': False, 'V_height': False,
'V_order': False, 'V_following': False, 'V_how': False, 'V_production': False, 'V_forward': False, 'V_below': False, 'V_t
hem': False, 'V_january': False, 'V_per': False, 'V_march': False, 'V_today': False, 'V_statements': False, 'V_best': Fal
```

```
# train a classifier and report accuracy
train_set, test_set = bigram_featuresets[100:], bigram_featuresets[:100]
classifier = nltk.NaiveBayesClassifier.train(train_set)
nltk.classify.accuracy(classifier, test_set)
```

```
7]: 0.95
```

The code provided splits the **bigram_featuresets** into training and testing sets, trains a Naive Bayes classifier on the training set, and then evaluates the accuracy of the classifier on the test set using the **nltk.classify.accuracy** function.

## IV. Experimenting with POS:

The below code defines a function named **POS_features** that takes a document (a list of words) and a list of word features as input. This function uses the default Part-of-Speech (POS) tagger, specifically the Stanford tagger, to assign POS tags to the words in the document. It then counts the occurrences of different POS tags and constructs a feature dictionary based on these counts.

By incorporating POS features in addition to the presence of specific words, the **POS_features** function provides a richer representation of the document. It captures information about the distribution of different POS tags, such as nouns, verbs, adjectives, and adverbs.

```
# this function takes a document list of words and returns a feature dictionary
# it runs the default pos tagger (the Stanford tagger) on the document
#   and counts 4 types of pos tags to use as features
def POS_features(document, word_features):
    document_words = set(document)
    tagged_words = nltk.pos_tag(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    numNoun = 0
    numVerb = 0
    numAdj = 0
    numAdverb = 0
    for (word, tag) in tagged_words:
        if tag.startswith('N'): numNoun += 1
        if tag.startswith('V'): numVerb += 1
        if tag.startswith('J'): numAdj += 1
        if tag.startswith('R'): numAdverb += 1
    features['nouns'] = numNoun
    features['verbs'] = numVerb
    features['adjectives'] = numAdj
    features['adverbs'] = numAdverb
    return features
```

```
# define feature sets using this function
POS_featuresets = [(POS_features(d, word_features), c) for (d, c) in res]
# number of features for document 0
print(len(POS_featuresets[0][0].keys()))
```

```
3004
```

```
# the first sentence
print(res[0])
# the pos tag features for this sentence
print('num nouns', POS_featuresets[0][0]['nouns'])
print('num verbs', POS_featuresets[0][0]['verbs'])
print('num adjectives', POS_featuresets[0][0]['adjectives'])
print('num adverbs', POS_featuresets[0][0]['adverbs'])
```

```
(['Subject', ':', 'i', 'put', 'in', '35000', 'for', 'the', 'expected', 'volume', 'for', 'the', 'sale', 'to', 'sds', '/', 'tu
fco', '.', 'sitara', '276366'], 'ham')
num nouns 6
num verbs 2
num adjectives 2
num adverbs 0
```

By accessing the POS tag features for a specific sentence, you can inspect the counts of different POS tags (nouns, verbs, adjectives, adverbs) in that sentence. This information can be useful for analyzing the linguistic characteristics of the text.

```
# train and test the classifier
train_set, test_set = POS_featuresets[100:], POS_featuresets[:100]
classifier = nltk.NaiveBayesClassifier.train(train_set)
nltk.classify.accuracy(classifier, test_set)
```

```
1]: 0.95
```

By running the classifier, we got the accuracy of 95%.

## V. Experimenting with Cross Validation

```
## 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_accuracy(num_folds, featuresets):
    subset_size = int(len(featuresets)/num_folds)
    print('Each fold size:', subset_size)
    accuracy_list = []
    # iterate over the folds
    for i in range(num_folds):
        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 (i, accuracy_this_round)
        accuracy_list.append(accuracy_this_round)
    # find mean accuracy over all rounds
    print ('mean accuracy', sum(accuracy_list) / num_folds)
```

The above code implements a cross-validation function to assess the accuracy of a classifier using a given number of folds. Here's a brief explanation of the code:

1. **cross_validation_accuracy(num_folds, featuresets)**: This function takes two arguments: **num_folds**, which represents the number of folds for cross-validation, and **featuresets**, which is a list of feature sets. Each feature set is a tuple containing a feature dictionary and a category label.
2. **subset_size = int(len(featuresets)/num_folds)**: Calculates the size of each fold by dividing the total number of feature sets by the number of folds. It determines how many feature sets should be included in each fold.
3. **print('Each fold size:', subset_size)**: Prints the size of each fold.
4. The function then proceeds to iterate over the folds using a **for** loop. For each iteration:
   a. **test_this_round** is assigned a subset of feature sets for testing, based on the current fold.
   b. **train_this_round** is assigned the remaining feature sets for training, excluding the ones used for testing in the current fold.
   c. The classifier (NaiveBayesClassifier in this case) is trained using **train_this_round**.
   d. The accuracy of the classifier is evaluated against the test set **test_this_round**, and the accuracy value is printed.
   e. The accuracy value for the current fold is appended to the **accuracy_list**.
5. After iterating over all folds, the function calculates and prints the mean accuracy by summing up all the accuracy values in **accuracy_list** and dividing by the number of folds.

This cross-validation procedure allows for a more reliable estimation of the classifier's performance by evaluating it on multiple subsets of the data.

- Running the normal feature sets and Bigram feature sets for different fold to experiment and check the accuracy percentage.

```
num_folds = 5
cross_validation_accuracy(num_folds, featuresets)
```

```
Each fold size: 400
0 0.96
1 0.98
2 0.96
3 0.9675
4 0.96
mean accuracy 0.9654999999999999
```

```
num_folds = 5
cross_validation_accuracy(num_folds, bigram_featuresets)
```

```
Each fold size: 400
0 0.96
1 0.98
2 0.96
3 0.9675
4 0.96
mean accuracy 0.9654999999999999
```

Trying the same with POS Tagged data.

### Comparing Cross Validation of POS_featuresets

```
num_folds = 5
cross_validation_accuracy(num_folds, POS_featuresets)
```

```
Each fold size: 400
0 0.9575
1 0.98
2 0.9625
3 0.97
4 0.9625
mean accuracy 0.9665000000000001
```

```
num_folds = 10
cross_validation_accuracy(num_folds, POS_featuresets)
```

```
Each fold size: 200
0 0.96
1 0.95
2 0.98
3 0.98
4 0.975
5 0.955
6 0.98
7 0.96
8 0.96
9 0.96
mean accuracy 0.966
```

POS tagged data clearly gives more accuracy, as the data is more accurately identified and tagged.

## VI. Precision, Recall and F1 Score:

### Precision, Recall and F1 Score

```
# Function to compute precision, recall and F1 for each label
#  and for any number of labels
# Input: list of gold labels, list of predicted labels (in same order)
# Output:  prints precision, recall and F1 for each label
def eval_measures(gold, predicted):
    # get a list of labels
    labels = list(set(gold))
    # these lists have values for each label
    recall_list = []
    precision_list = []
    F1_list = []
    for lab in labels:
        # for each label, compare gold and predicted lists and compute values
        TP = FP = FN = TN = 0
        for i, val in enumerate(gold):
            if val == lab and predicted[i] == lab:  TP += 1
            if val == lab and predicted[i] != lab:  FN += 1
            if val != lab and predicted[i] == lab:  FP += 1
            if val != lab and predicted[i] != lab:  TN += 1
        # use these to compute recall, precision, F1
        recall = TP / (TP + FP)
        precision = TP / (TP + FN)
        recall_list.append(recall)
        precision_list.append(precision)
        F1_list.append( 2 * (recall * precision) / (recall + precision))

    # the evaluation measures in a table with one row per label
    print('\tPrecision\tRecall\t\tF1')
    # print measures for each label
    for i, lab in enumerate(labels):
        print(lab, '\t', "{:10.3f}".format(precision_list[i]), \
          "{:10.3f}".format(recall_list[i]), "{:10.3f}".format(F1_list[i]))
```

The above code implements a function called **eval_measures** to compute precision, recall, and F1 scores for each label in a classification task. Here's a brief explanation of the code:

1. **eval_measures(gold, predicted)**: This function takes two arguments: **gold**, which is a list of gold labels (ground truth), and **predicted**, which is a list of predicted labels from a classifier (in the same order as the gold labels).
2. **labels = list(set(gold))**: Creates a list of unique labels present in the gold labels.
3. The function initializes empty lists for **recall_list**, **precision_list**, and **F1_list**, which will store the evaluation measures for each label.
4. The function then iterates over each label in **labels**. For each label:
   a. Initializes the variables TP (True Positives), FP (False Positives), FN (False Negatives), and TN (True Negatives) to 0.

b. Iterates over the elements in the gold and predicted lists simultaneously using **enumerate()**.
c. Compares the gold and predicted labels at each position. If they match or don't match, the corresponding counters (TP, FP, FN, TN) are incremented accordingly.
d. Computes recall, precision, and F1 scores based on the TP, FP, and FN counts.
e. Appends the computed values to **recall_list**, **precision_list**, and **F1_list** for the current label.

5. After iterating over all labels, the function prints a table showing the precision, recall, and F1 score for each label.

The **eval_measures** function is useful for evaluating the performance of a classifier by providing a detailed breakdown of evaluation measures for each label.

```
# call the function with our data
eval_measures(goldlist, predictedlist)
```

| | Precision | Recall | F1 |
|---|---|---|---|
| spam | 0.980 | 0.923 | 0.950 |
| ham | 0.922 | 0.979 | 0.949 |

**F1 scores represent the model's accuracy for the data sets, and here we can see that ham and spam have 95.0 & 94.9 accuracy, which is good.**

```
from nltk.classify import SklearnClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC, NuSVC
```

```
#support vector classification
SVC_classifier = SklearnClassifier(SVC())
SVC_classifier.train(train_set)
print("SVC_classifier accuracy percent:", (nltk.classify.accuracy(SVC_classifier, test_set))*100)
```

```
SVC_classifier accuracy percent: 68.0
```

```
#kernal parameter is set to linear
LinearSVC_classifier = SklearnClassifier(LinearSVC())
LinearSVC_classifier.train(train_set)
print("LinearSVC_classifier accuracy percent:", (nltk.classify.accuracy(LinearSVC_classifier, test_set))*100)
```

```
C:\Users\udayv\anaconda3\lib\site-packages\sklearn\svm\_base.py:1244: ConvergenceWarning: Liblinear failed to converge, incr
ease the number of iterations.
  warnings.warn(
```

```
LinearSVC_classifier accuracy percent: 96.0
```

```
#SVC with lower and upper bounds for margin error
NuSVC_classifier = SklearnClassifier(NuSVC())
NuSVC_classifier.train(train_set)
print("NuSVC_classifier accuracy percent:", (nltk.classify.accuracy(NuSVC_classifier, test_set))*100)
```

```
NuSVC_classifier accuracy percent: 91.0
```

**The above results are with more methods and libraries not discussed in class.**

3 functions from the sklearn library are used: SVC, Linear SVC & NuSVC. Linear one gives us the highest accuracy of 96%.

1. **SVC** (Support Vector Classification): **SVC** is a class that represents the Support Vector Machine classifier for both binary and multiclass classification. It uses the C-Support Vector Classification formulation and can handle non-linear decision boundaries through the use of kernel functions.

2. **LinearSVC** (Linear Support Vector Classification): **LinearSVC** is a variant of SVM that uses a linear kernel function for classification. It is optimized for linearly separable data and works well in high-dimensional spaces. Unlike **SVC**, **LinearSVC** does not support kernel functions, but it can handle large-scale datasets more efficiently.

3. **NuSVC** (Nu-Support Vector Classification): **NuSVC** is another variant of SVM that uses a parameter called "nu" for controlling the number of support vectors. It allows for more flexible control over the trade-off between the number of support vectors and the complexity of the decision boundary. Like **SVC**, it supports kernel functions and can handle both binary and multiclass classification problems.

**Summary:**

1. **Processing the text and Tokenization:**

   In this step I have preprocessed the text using required functions and implemented tokenization which means to split it into individual words, and the tokenized text along with its label ('spam' or 'ham').

   I have also calculated the frequency distribution of words in a list using the **nltk.FreqDist()** function.

   I have also splitted the **featuresets** into training and test sets, trains a Naive Bayes classifier on the training set, and evaluated the accuracy of the classifier on the test set.

2. **Filtering By Stop Words**

   I have also initialized a list of stop words, added additional stopwords, created a new list of stopwords excluding the additional ones, filtered the original word list

by removing the stopwords, and calculated the frequency distribution of the filtered words.

3. **Experimenting with BiGraphs:**

   I have imported the **nltk.collocations** module and specifically imported the **BigramAssocMeasures** class from it.

   I have splitted the **bigram_featuresets** into training and testing sets, trained a Naive Bayes classifier on the training set, and then evaluates the accuracy of the classifier on the test set using the **nltk.classify.accuracy** function.

4. **Experimenting with POS:**

   I have defined a function named **POS_features** that takes a document (a list of words) and a list of word features as input. This function uses the default Part-of-Speech (POS) tagger, specifically the Stanford tagger, to assign POS tags to the words in the document.

   By accessing the POS tag features for a specific sentence, I have inspected the counts of different POS tags (nouns, verbs, adjectives, adverbs) in that sentence.

   Finally after running the classifier, I have received accuracy of 95%.

5. **Experimenting with Cross Validation:**

   In this section I have experimented with cross validations on features, bi gram features and POS_features based on number of folds.

   I have observed that the accuracy of POS_features is more when compared to other features.

6. **Precision, Recall and F1 Score:**

   I have implemented a function called **eval_measures** to compute precision, recall, and F1 scores for each label in a classification task.
   The results are as follows.

```
# call the function with our data
eval_measures(goldlist, predictedlist)
```

```
        Precision    Recall        F1
spam        0.980     0.923     0.950
ham         0.922     0.979     0.949
```

7. **Comparing results from different models:**

   By comparing the models of SVC, Linear SVC, NuSVC I have obtained greater results with Linear SVC model with accuracy of 96%.