Classifiers used:

1. NaiveBayes
   1. BernoulliNB
   2. MultinomialNB
2. XGBClassifier
3. SVC

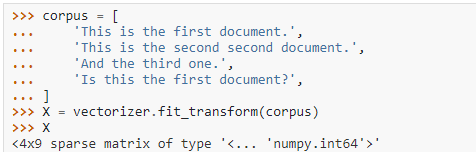
Test Feature extraction/BagOfWords/Vectorization:

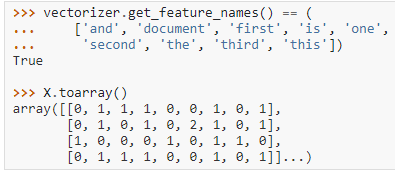
Raw data cannot be fed directly to the algorithms themselves as most of them expect numerical feature vectors with a fixed size rather than the raw text documents with variable length. Most common ways to extract numerical features from text content,

1. tokenizing strings and giving an integer id for each possible token, for instance by using white-spaces and punctuation as token separators.
2. counting the occurrences of tokens in each document.
3. normalizing and weighting with diminishing importance tokens that occur in the majority of samples / documents.

Vectorizers used:

1. CountVectorizer() - implements both tokenization and occurrence counting in a single class.





1. TfidfVectorizer()

In a large text corpus, some words like "the", "a","is” might be occuring very frequently but they dont really carry useful information that can be used for classification. If we were to feed the direct count data directly to a classifier those very frequent terms would shadow the frequencies of rarer yet more interesting terms. Tf means term-frequency while tf–idf means term-frequency times inverse document-frequency. The goal of using tf-idf instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus. This is done by TfidTransformer.

TfidVectorizer is a equivalent of CountVectorizer followed by TfidfTransformer.

1. HashingVectorizer()

The above vectorization scheme is simple but the fact that it holds an in- memory mapping from the string tokens to the integer feature indices (the vocabulary\_ attribute) causes several problems when dealing with large datasets:

* The larger the corpus, the larger the vocabulary will grow and hence the memory use too

It turns a collection of text documents into a scipy.sparse matrix holding token occurrence counts (or binary occurrence information), possibly normalized as token frequencies if norm=’l1’ or projected on the euclidean unit sphere if norm=’l2’. This text vectorizer implementation uses the hashing trick to find the token string name to feature integer index mapping.

Other preprocessing:

1. Removed all punctuations and additional spaces in the input text
2. Stemmer from the NLTK library for clubbing similar words together
3. Converted every message to lowercase
4. Removed stopwords from the text (nltk.corpus). Stop words are words like “and”, “the”, “him”, which are presumed to be uninformative in representing the content of a text.

Results obtained:

Sample output:

Running each classifier...

------------------------------------------------------------------

For classifier: bNB , vectorizer: cVec and parameters: {'bNB\_\_alpha': [0.05, 0.1, 0.5, 1.0, 1.2]}

Best Parameters: {'bNB\_\_alpha': 0.05}

Confusion Matrix:

Predicted HAM(0) Predicted SPAM(1)

Actual HAM(0) 965 0

Actual SPAM(1) 13 137

Scores for the Test data >>> Accuracy: 0.98834 , f1 score: 0.9547 , Precision: 1.0 , ROC\_AUC: 0.95667

------------------------------------------------------------------

**Conclusion:**

As it can be seen from the below table, Naïve Bayes with countVectorizer or TfidfVectorizer seems to be the best when Naïve Bayes, SVC and XGBoost are compared with each other.

Naïve Bayes with countVectorizer or TfidfVectorizer > SVC > XGBoost

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **BernoulliNB** | | | **MultinomialNB** | | | **XGBClassifier** | | | **SVC** | | | **LinearSVC** | | | **NB Implementaion** |
| cVect | tfVect | hVect | cVect | tfVect | hVect | cVect | tfVect | hVect | cVect | tfVect | hVect | cVect | tfVect | hVect |
| **Accuracy** | 0.989 | 0.989 | 0.865 | 0.984 | 0.984 | 0.966 | 0.966 | 0.969 | NA | 0.981 | 0.981 | 0.976 | 0.982 | 0.983 | 0.978 | 0.974 |
| **f1 score** | 0.958 | 0.958 | 0.000 | 0.938 | 0.937 | 0.858 | 0.858 | 0.868 | NA | 0.925 | 0.927 | 0.905 | 0.930 | 0.933 | 0.911 | 0.905 |
| **Precision** | 1.000 | 1.000 | 0.000 | 0.971 | 0.993 | 0.975 | 0.975 | 1.000 | NA | 0.992 | 0.971 | 0.956 | 0.985 | 0.985 | 0.977 | 0.890 |
| **ROC\_AUC** | 0.960 | 0.960 | 0.500 | 0.951 | 0.943 | 0.882 | 0.882 | 0.883 | NA | 0.933 | 0.941 | 0.927 | 0.939 | 0.942 | 0.925 | 0.951 |

Other observations:

1. When SnowballStemmer (for clubbing similar words together) functionality is used the accuracy, f1 score, precision and ROC\_AUC scores are little lower than the case when its not used.
2. From EDA we can see spam messages are longer than ham messages, concentrated around 150 characters. But adding additional column(data['msg\_length']) to the input data to capture the message length doesn’t improve the scores, it remains same as before.