Spam Mail Classification

CS550 - Machine Learning

Project Report submitted by
Swati Kumari (12010150)
Harshit Kumar Choudhary (12010260)

1 Abstract

The increasing volumes of unsolicited bulk e-mail (also known as spam) are bringing more annoyance for most Internet users. Using a classifier based on a specific machine learning technique to automatically filter out spam e-mail has drawn many researchers' attention. In this project, we implement some of the most popular machine learning methods, such as, Naive Bayes, Decision Tree, Support Vector Machine (SVM) and Random Forest. We analyse their applicability to the problem of spam email classification using different performance metrics.

2 Introduction

In recent years, e-mails have become a common and important medium of communication for most Internet users. However, spam, also known as unsolicited commercial/ bulk e-mail, is a bane of e-mail communication. A study estimated that over 70% of today's business e-mails are spam. Therefore, there are many serious problems associated with growing volumes of spam such as filling user's mailboxes, engulfing important personal mail, wasting storage space and communication bandwidth, and consuming user's time to delete all spam mails. Spam mails vary significantly in content and they roughly belong to the following categories: money making scams, fat loss, improve business, make friends, service provider advertisement, etc. Automatic e-mail filtering seems to be the most effective method for countering spam at the moment and a tight competition between spammers and spam-filtering methods is going on. Only several years ago most of the spam could be reliably dealt with by blocking e-mails coming from certain addresses or filtering out messages with certain subject lines. Spammers began to use several tricky methods to overcome the filtering methods like using random sender addresses and/or append random characters to the beginning or the end of the message subject line. Knowledge engineering and machine learning are the two general approaches used in e-mail

filtering. In knowledge engineering approach a set of rules has to be specified according to which emails are categorized as spam or ham. A set of such rules should be created either by the user of the filter, or by some other authority (e.g. the software company that provides a particular rule-based spam-filtering tool). By applying this method, no promising results shows because the rules must be constantly updated and maintained, which is a waste of time and it is not convenient for most users. Machine learning approach is more efficient than knowledge engineering approach; it does not require specifying any rules [4]. Instead, a set of training samples, these samples is a set of pre-classified e-mail messages. A specific algorithm is then used to learn the classification rules from these e-mail messages. Machine learning approach has been widely studied and there are lots of algorithms can be used in e-mail filtering. They include Naïve Bayes, Support Vector Machines, Neural Networks, K-nearest neighbour, Decision Tree, Random Forest, etc. From the machine learning view point, spam filtering based on the textual content of e-mail can be viewed as a special case of text categorization, with the categories being spam or non-spam (ham). In this project, we evaluate the performance of four machine learning algorithms: Naive Bayes, Support Vector Machines, Decision Tree and Random Forest.

3 Problem Definition

In this project, we classify e-mails into spam and ham using four machine learning models, Naive Bayes, Support Vector Machines, Decision Tree and Random Forest. We analyse the performance of these models based on several parameters.

4 Objective

To achieve our goal to classify e-mails into spam and ham, we first we need to prepare the data and then train a classifier model, which will classify our mails. To prepare the data, we followed the steps below:

- Download spam and ham emails through Google's takeout service as an mbox file from our personal email id.
- Read the mbox files into lists using the 'mailbox' package. Each element in the list contains an individual email.
- Unpack each email and concatenate their subject and body.
- Convert the lists to dataframes, join the spam and ham dataframes and shuffle the resultant dataframe.
- Split the dataframe into train and test dataframes. The test data should contain 33% of the original dataset.

- Split the mail text into lemmas and apply TF-IDF transformation using CountVectorizer followed by TF-IDF transformer.
- Using the trained models, we can predict the correctness of our models using various performance metrics.

5 Models

Different models used in our project are:

1. **Naive Bayes:** The Naive Bayes (NB) classifier is a probability-based approach. The basic concept of it is to find whether an e-mail is spam or not by looking at which words are found in the message and which words are absent from it. This approach begins by studying the content of a large collection of e-mails which have already been classified as spam or legitimate. Then when a new e-mail comes into some user's mailbox, the information gleaned from the training set is used to compute the probability that the e-mail is spam or not given the words appearing in the e-mail.

   Given a feature vector $\overrightarrow{x} = (x_1, x_2, \cdots, x_n)$ of an e-mail, where are the values of attributes $X_1, \cdots, X_n$, and n is the number of attributes in the corpus. Here, each attribute can be viewed as a particular word occurring or not. Let c denote the category to be predicted, i.e., $c \in (spam, ham)$, by Bayes law the probability that $\overrightarrow{x}$ belongs to c is as given in

   $$P(c|\overrightarrow{x}) = \frac{P(c).P(\overrightarrow{x}|c)}{P(\overrightarrow{x})} \tag{1}$$

   where $\overrightarrow{x}$ denotes the a-priori probability of a randomly picked e-mail has vector $\overrightarrow{x}$ as its representation, P(c) is also the a prior probability of class c (that is, the probability that a randomly picked e-mail is from that class), and $P(\overrightarrow{x}|c)$ denotes the probability of a randomly picked e-mail with class c has $\overrightarrow{x}$ as its representation. The probability $P(\overrightarrow{x}|c)$ is almost impossible to calculate because the fact that the number of possible vectors $\overrightarrow{x}$ is too high. In order to alleviate this problem, it is common to make the assumption that the components of the vector $\overrightarrow{x}$ are independent in the class. Thus, $P(\overrightarrow{x}|c)$ can be decomposed to:

   $$P(\overrightarrow{x}|c) = \prod_{i=1}^{n} P(x_i|c) \tag{2}$$

   So, using the NB classifier for spam filtering can be computed as

   $$C_{NB} = argmax_{c \in (spam,ham)} P(c) \prod_{i=1}^{n} P(x_i|c) \tag{3}$$

2. **Support vector machine:** Support vector machine (SVM) is a new and very popular technique for data classification in the machine learning community. SVM has been shown to be very effective in the field of text categorization because it has the ability to handle high-dimensional data by using kernels. When using SVM for pattern classification, the basic idea is to find the optimal separating hyperplane that gives the maximum margin between the positive and negative samples. According to the idea, the spam filtering can be viewed as the simple possible SVM application: classification of linearly separable classes; that is, a new e-mail either belongs or does not to the spam category.

   Given a set of training samples $X = (x_i, y_i)$, where $x_i \in R^m$ and $y_i \in (+1, -1)$ is the corresponding output for the i^{th} training sample (here +1 represents spam and -1 stands for ham). The output of a linear SVM is

$$y = w.x - b \tag{4}$$

   where y is the result of classification, w is the normal weight vector corresponding to those in the feature vector x, and b is the bias parameter in the SVM model that determined by the training process. Maximizing the margin can be achieved through the following optimization problem:
   minimize

$$\frac{1}{2}||w||^2 \tag{5}$$

   subject to

$$y_i(w.x + b) \geq 1, \forall i \tag{6}$$

3. **Decision Tree:** A decision tree is a kind of machine learning algorithm that can be used for classification or regression. It is a simple flowchart that selects labels for input values. This flowchart consists of decision nodes, which check feature values, and leaf nodes, which assign labels. To choose the label for an input value, we begin at the flowchart's initial decision node, known as its root node. This node contains a condition that checks one of the input value's features, and selects a branch based on that feature's value. Following the branch that describes our input value, we arrive at a new decision node, with a new condition on the input value's features. We continue following the branch selected by each node's condition, until we arrive at a leaf node which provides a label for the input value. Once we have a decision tree, it is straightforward to use it to assign labels to new input values. What's less straightforward is how we can build a decision tree that models a given training set. There are four points we have to consider:

   - How many splits per node?
   - Which dimension do we test?
   - When do we stop?
   - How do we assign class labels to the leaf nodes?

Ideally, we want to choose values so that the resulting split causes one class to be much more present than the others. When we only have training data, at each node, we'll be keeping track of how many examples of each class are present at a node as a vector. We want to make splits so that we have many examples of only one particular class and few examples of the other. In other words, we want the split that decreases the entropy. High entropy means that we have a mix of different classes; low entropy means that we have predominantly one class. Ideally, we want our nodes to have no entropy, i.e., all examples at this node are definitely of one class. This low entropy is desirable at the leaf nodes since, when we classify an example, we can be very sure of its class in a low entropy leaf node.
Mathematically, there are several ways to measure this. One way is using the definition of entropy.

$$S(N) = -\sum_i p_i \log_2 p_i \tag{7}$$

This is computing the entropy at node N by summing over all classes i and computing $p_i \log_2 p_i$ where p_i is the proportion of examples that belong to class i at node N.
There is another, more commonly used metric called Gini impurity. Impurity and entropy mean the same thing: we want lower Gini impurity. We can compute the Gini impurity using the following.

$$G(N) = \sum_i p_i(1 - p_i) \tag{8}$$

Using a decision tree, we can keep splitting until each leaf node only has a single training example. Ideally, at the leaf node, we'll have zero entropy/impurity, and we simply select the only available class. However, in many cases, we won't be that lucky. We might have a few examples from each class. There are several ways of handling this. A common thing to do is to randomly sample from the resulting distribution at that leaf node. We can also consider the total number of each class.

4. **Random Forest:** Random forest is a supervised learning algorithm which is used for both classification as well as regression. But however, it is mainly used for classification problems. As we know that a forest is made up of trees and more trees means more robust forest. Similarly, random forest algorithm creates decision trees on data samples and then gets the prediction from each of them and finally selects the best solution by means of voting. It is an ensemble method which is better than a single decision tree because it reduces the overfitting by averaging the result. It overcomes the problem of overfitting by averaging or combining the results of different decision trees. We can understand the working of Random Forest algorithm with the help of following steps:

    - Step 1: First, start with the selection of random samples from a given dataset.

- Step 2: Next, this algorithm will construct a decision tree for every sample. Then it will get the prediction result from every decision tree.
- Step 3: In this step, voting will be performed for every predicted result.
- Step 4: At last, select the most voted prediction result as the final prediction result.

6 Datasets Used

The datasets are the emails which are downloaded from the the google takeout of the personal mail ids and the file format is **mbox**. For spam we have few sets and for ham we have another sets of mails. A special type of package **mailbox** is used to be imported to deal with mbox files.

7 Problems Faced

The first problem is number of spam emails are not much available on the mail ids as it automatically gets deleted after certain period. The second problem is selecting better parameters of the model to get better performance.

8 Implementation

Implementation has been done in three phases, preprocessing of mail text, training the model and measuring the performance of each model used, using various performance metrices, which are shown in section 9.1. All the phases are descibed below.

8.1 Preprocessing

In actual mail data are semi-unstructured in nature, and so it is difficult to handle these kind of data. First we need to preprocess these data in order to make structured and to get valuable texts and features to proceed further. The steps are as follows:-

- **Reading mbox file**: As mail data is of file format **.mbox**, so we need special package 'mailbox' to access the email text. Using the following python code we can read mbox file: **message.get-payload(decode=True)**.
- **Stop-word Removal**: Stop-word removal involves removing frequently used non-informative words, e.g. 'a', 'an', 'the', and 'is', etc.
- **Lemmatization:** Word-lemmatization is a term used to describe a process of converting words to their morphological base forms, mainly eliminating plurals, tenses, gerund forms, prefixes and suffixes. Lemmatization,

while reducing a word considers the part of speech and the context of the word. The primary advantages of employing lemmatization are feature space dimension reduction and classifier accuracy.

- **Bag of Words (BOW):** After lematization, texts need to be converted into some machine readable inputs. As we know machine can be feeded with numbers only. Bag of words is a transformation of text documents into vectors of numbers. In bag of words (BOW) model (or vector-space model), words occurring in the e-mail are treated as features. Given a set of terms $T = (t_1, t_2, t_3, \cdots, t_n)$, the bag of words model represents a document d as an N-dimensional feature vector $x = (x_1, x_2, x_3, \cdots, x_n)$ where x_i is a function of the occurrence of t_i in d.

- **Term Frequency-Inverse Document Frequency (TF-IDF):** TF-IDF is a statistical measure that evaluates how relevant a word is to a document in a collection of documents. This is done by multiplying two metrics: how many times a word appears in a document, and the inverse document frequency of the word across a set of documents. It has many uses, most importantly in automated text analysis, and is very useful for scoring words in machine learning algorithms for Natural Language Processing (NLP).
  TF-IDF score for the word t in the document d from the document set D is calculated as follows:

$$tfidf(t, d, D) = tf(t, d).idf(t, D) \tag{9}$$

  where

$$tf(t, d) = log(1 + freq(t, d)) \tag{10}$$

$$idf(t, D) = log\frac{N}{count(d \in D : t \in d)} \tag{11}$$

8.2 Training the model

We have now training data(train-features) and test data(test-features) to train the models. Four machine learning models are used in this project.

- **Multinomial Naive Bayes Classifier:** The model is used with the shown parameters. **MultinomialNB(alpha=0, class_prior=None, fit_prior=True)**

  alpha=0, alpha is smoothing factor, 0 indicates that training doesn't need any smoothing.
  class_prior=None, if there is any prior probabilities of classes are given.
  fit_prior=True, fit the model with prior probabilities of classes are given.

- **Decision Tree Classifier:** This model is used with default parameters as shown in below function:

DecisionTreeClassifier(ccp _alpha=0.0, class_weight=None, criterion='gini', max_depth=None, max_features=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, presort='deprecated', random_state=None, splitter='best')

ccp_alpha: non-negative float, default=0.0 Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen.
class_weight: dict, list of dict or "balanced", default=None, Weights associated with classes in the form class_label: weight. If None, all classes are supposed to have weight one.

criterion{"gini", "entropy"}, default="gini" The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

max_depth: int, default=None, The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

max_features: int, float or {"auto", "sqrt", "log2"}, default=None, The number of features to consider when looking for the best split: If None, then max_features=n_features.

max_leaf_nodes: int, default=None, Grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease: float, default=0.0, A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

min_impurity_split: float, default=0 Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

min_samples_leaf: int or float, default=1 The minimum number of samples required to be at a leaf node.

min_weight_fraction_leaf float, default=0.0 The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

random_state: int, RandomState instance or None, default=None, Controls the randomness of the estimator.

splitter{"best", "random"}: default="best" The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

- **Support Vector Classifier:** This model is also used with default parameters as

SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf', max_iter=-1, probability=False, random_state=None, shrinking=True, tol=0.001, verbose=False)

C: float, default=1.0, Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

kernel{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, default='rbf',

Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape (n_samples, n_samples).

degree: int, default=3, Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma{'scale', 'auto'} or float, default='scale' Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. If gamma='scale' (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma, if 'auto', uses 1 / n_features.

coef0: float, default=0.0 Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking: bool, default=True, Whether to use the shrinking heuristic. See the User Guide.

probability: bool, default=False, Whether to enable probability estimates. This must be enabled prior to calling fit, will slow down that method as it internally uses 5-fold cross-validation, and predict_proba may be inconsistent with predict. Read more in the User Guide.

tol: float, default=1e-3, Tolerance for stopping criterion.

cache_size: float, default=200, Specify the size of the kernel cache (in MB).

class_weight: dict or 'balanced', default=None Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y))

verbose: bool, default=False, Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter int, default=-1, Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape{'ovo', 'ovr'}, default='ovr', Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). However, one-vs-one ('ovo') is always used as multi-class strategy. The parameter is ignored for binary classification.

break_ties: bool, default=False, If true, decision_function_shape='ovr', and number of classes ¿ 2, predict will break ties according to the confidence values of decision_function; otherwise the first class among the tied classes is returned. Please note that breaking ties comes at a relatively high computational cost compared to a simple predict.

random_state: int, RandomState instance or None, default=None, Controls the pseudo random number generation for shuffling the data for probability estimates. Ignored when probability is False. Pass an int for reproducible output across multiple function calls. See Glossary.

- **Random Forest Classifier:** RF model is used here with few selected parameters as:
  RandomForestClassifier(n_estimators=20,criterion='entropy')

  n_estimatorsint:, default=100, The number of trees in the forest. Here 20 is taken as 20 trees are enough to train our data.

  criterion{"gini", "entropy"}:, default="gini", The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. This parameter is tree-specific. Here entropy is selected as we have to gain information only not to deal with impurity.

9 Result And Performance

9.1 Performance Metrics

Various performance metrics used to analyse the performance of different models are:

- **Confusion Matrix:** It is the easiest way to measure the performance of a classification problem where the output can be of two or more type of classes. A confusion matrix is a table with two dimensions viz. "Actual" and "Predicted" and furthermore, both the dimensions have "True Positives (TP)", "True Negatives (TN)", "False Positives (FP)", "False Negatives (FN)".

  Explanation of the terms associated with confusion matrix are as follows:

  True Positives (TP): It is the case when both actual class and predicted class of data point is 1.
  True Negatives (TN): It is the case when both actual class and predicted class of data point is 0.
  False Positives (FP): It is the case when actual class of data point is 0 and predicted class of data point is 1.
  False Negatives (FN): It is the case when actual class of data point is 1 and predicted class of data point is 0.

- **Accuracy**: It is most common performance metric for classification algorithms. It may be defined as the number of correct predictions made as a ratio of all predictions made. We can easily calculate it by confusion matrix with the help of following formula:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{12}$$

- **Precision:** Precision, used in document retrievals, may be defined as the number of correct documents returned by our ML model. We can easily calculate it by confusion matrix with the help of following formula:

$$Precision = \frac{TP}{TP + FP} \tag{13}$$

- **Recall:** Recall may be defined as the number of positives returned by our ML model. We can easily calculate it by confusion matrix with the help of following formula:

$$Recall = \frac{TP}{TP + FN} \tag{14}$$

- **F-score:** This score will give us the harmonic mean of precision and recall. Mathematically, F1 score is the weighted average of the precision and recall. The best value of F1 would be 1 and worst would be 0. We can calculate F1 score with the help of following formula:

$$F1 = 2(Precision * Recall)/(Precision + Recall) \tag{15}$$

- **AUC:** AUC (Area Under Curve)- ROC (Receiver Operating Characteristic) is a performance metric, based on varying threshold values, for classification problems. ROC is a probability curve and AUC measure the separability. AUC-ROC metric will tell us about the capability of model in distinguishing the classes. Higher the AUC, better the model.

9.2 Results

We have trained the machine using 4 different models and now time to check their performances using above discussed performance metrices. We get the results shown in below table. As we can observe all models are performing well on our datasets, but the best one is Random forest classifier with 93 percent accuracy.

Results					
Models ↓ Parameters →	Accuracy	Precision	Recall	F-score	AUC
Naive Bayes	0.90	0.8	0.63	0.70	0.79
SVM	0.90	0.8	0.63	0.70	0.79
Random Forest	0.93	0.80	0.86	0.83	0.91
Decision Tree	0.86	0.58	0.89	0.70	0.87

10 Conclusion And Further Work

The detection of spam e-mail is an important issue of information technologies, and machine learning algorithms play a central role in this topic. In this project, we analysed the performance of four different machine learning model in classifying e-mails as spam and ham. Our implementation showed that Random Forest worked best for the given classification problem, while Decision Tree's accuracy was the least among the four models.

In future, we plan about getting insights from email. A tough task will be automated for the companies and organizations to get the insight from large number of emails. The reader won't need to go through all the emails one by one to find out the sentiments, so that companies can take quick action to resolve the issues or improve the quality as per the demands.

11 Reference

Source for code:
https://github.com/nissim-panchpor/Email-Spam-Detecting-ML-Classifier/blob/master/Email