**Feature Ranking and Preprocessing**

From the given feature vectors of total 58 features, I use feature ranking to select the subset of features and decrease the dimensionality. I rank the given set of features using the following approach:

1. Separate the feature vectors for spam and non-spam emails into spam sets and non-spam sets.
2. Calculate the mean for each feature vector in the spam set.
3. Calculate the mean for each feature vector in the non-spam set.
4. Find the deference of the mean for each feature vector.
5. Use a threshold (-0.002 and 0.009) to select the features differ a lot for spam and non-spam emails

From the deference of the mean, one can infer that some input features have high value for spam while others for non-spam. (if the difference gives the higher value). Other features show less information and they are appropriate to be eliminated.

Based on the above threshold, the chosen algorithm is applied on the reduced datasets and gives the better accuracy.

Using no data preprocessing in advanced, the accuracy of MLP algorithm was about 94 percent. So, the second choice was Naïve Bayes classifier for multivariate Bernoulli models. It is suitable for discrete data but it is designed to works with Binary/Boolean features. By changing its *binarize* parameter to 0.1 the accuracy of this algorithm increased to 96. But it made no sense to use this algorithm working with the given dataset. Finally, by applying the above feature ranking procedure and the following preprocessors from *scikit* *learn* library, the accuracy of 98 percent is achieved:

features = preprocessing.scale(features)

for training the algorithm by 10-fold cross validation and:

scaler = preprocessing.StandardScaler().fit(trainFeatures)

trainFeatures = scaler.transform(trainFeatures)

testFeatures = scaler.transform(testFeatures)

For training and testing without 10-fold cross validation. which standardize features by removing the mean and scaling to unite variance. The first two lines were used to standardize the training data. So, the scaler can then be used on new data to transform it the same way it did on the training set

**Multilayer Perceptron (MLP)- Classifier**

Based on the assumption that the given dataset has enough number of the training instances, I have chosen the MLP classifier. A multilayer perceptron is a feed forward artificial neural network model that maps sets of input data onto a set of appropriate output. It uses a back propagation algorithm which is a standard algorithm for any supervised-learning pattern recognition process. The training of the perceptron is performed using an iterative method, where the weight and bias vectors are initialized. To ensure the classification of a new occurrence training sample is adjusted for each iteration.

For instance let *x* be a vector that the perceptron fails to classify, and wi, bi the vector of weight and bias which corresponds to the ith iteration. We have sign(wix+bn) ≠ c where c is the sign corresponding to the real class of the message that has the characteristic vector x. The new vectors wi+1 and bi+1 are calculated as follows:

wi+1 = wi + cx and bi+1 = bi + c

The training continues until the perceptron manages to classify correctly all the feature vectors of the training sample.

The reason I had chosen this algorithm was its ability to solve problems stochastically, which often allows approximate solutions for extremely complex problems like fitness approximation. It can also be used to create mathematical models by regression analysis. As classification is a particular case of regression, when the response variable is categorical, MLP makes good classifier algorithm.

For spam classification by MLP classifier, there is a class in *scikit* *learn*, which is a free software machine learning library for the Python programming language, is used in this project. This algorithm and some parameters which are used in this project is shown as follows:

|  |
| --- |
| MLPClassifier(hidden\_layer\_sizes=(100, ), activation=’relu’, solver=’adam’, alpha=0.0001, batch\_size=’auto’, learning\_rate=’constant’, learning\_rate\_init=0.001, power\_t=0.5, max\_iter=200, shuffle=True, random\_state=None, tol=0.0001, verbose=False, warm\_start=False, momentum=0.9, nesterovs\_momentum=True, early\_stopping=False, validation\_fraction=0.1, beta\_1=0.9, beta\_2=0.999, epsilon=1e-08) |
| hidden\_layer\_sizes : tuple, length = n\_layers - 2, default (100,)  The ith element represents the number of neurons in the ith hidden layer. |
| hidden\_layer\_sizes : tuple, length = n\_layers - 2, default (100,)  The ith element represents the number of neurons in the ith hidden layer. |
| activation : {‘identity’, ‘logistic’, ‘tanh’, ‘relu’}, default ‘relu’  Activation function for the hidden layer.   * ‘identity’, no-op activation, useful to implement linear bottleneck, returns f(x) = x * ‘logistic’, the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)). * ‘tanh’, the hyperbolic tan function, returns f(x) = tanh(x). * ‘relu’, the rectified linear unit function, returns f(x) = max(0, x) |
| solver : {‘lbfgs’, ‘sgd’, ‘adam’}, default ‘adam’  The solver for weight optimization.   * ‘lbfgs’ is an optimizer in the family of quasi-Newton methods. * ‘sgd’ refers to stochastic gradient descent. * ‘adam’ refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba |
| learning\_rate\_init : 0.001 as a default  The initial learning rate used. |
| max\_iter : int, optional, default 200  Maximum number of iterations. The solver iterates until convergence (determined by ‘tol’) or this number of iterations. For stochastic solvers (‘sgd’, ‘adam’), note that this determines the number of epochs (how many times each data point will be used), not the number of gradient steps. |
| shuffle : bool, optional, default True  Whether to shuffle samples in each iteration. Only used when solver=’sgd’ or ‘adam’. |
| tol : float, optional, default 1e-4  Tolerance for the optimization. When the loss or score is not improving by at least tol for two consecutive iterations, unless learning\_rate is set to ‘adaptive’, convergence is considered to be reached and training stops. |
| warm\_start : bool, optional, default False  When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. |

The number of neurons in the hidden layer is chosen by trial and error. It was basically chosen according to the number of input features, while the best result was achieved when two layers with 45 neurons in the first and 7 neurons in the second layer had been selected.

Though the logistic sigmoid has a nice biological interpretation, it turns out that the logistic sigmoid can cause a neural network to get stuck during training. This id due in part to the fact that if a strongly-negative input, since I use preprocessor on dataset, is provided to the logistic sigmoid, it outputs values very near zero. Since neural network use the feed forward activations to calculate parameter gradients, this can result in model parameters that are updated less regularly than we would like, and are thus stuck in their current state. So, *tanh* is selected as an activation function is tanh. An alternative to the logistic sigmoid is the hyperbolic tangent:

*adam* is selected as a solver in this algorithm where it works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, ‘lbfgs’ can converge faster and perform better. It is an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients.

Since the evaluator and the main function shuffles the datasets in advanced, the shuffle parameter gets False in this algorithm.

**Area Under the Receiver Operating Characteristic Curve (AUROC)**

For spam filtering the best metrics can optimize it for precision or specificity. Roc curve can help us to choose a threshold that balances sensitivity and specificity in a way that makes sense for our particular context.

AUC is the percentage of the ROC plot that is underneath the curve. The higher the value AUC generates, the better the classifier is. If you randomly choose one positive and one negative observation, AUC represents the likelihood that your classifier will assign a higher predicted probability to the positive observation. It is even useful when there is high class imbalance.

In conclusion using AUC/ROC helps you not set a classification threshold and still useful when there is high class imbalance. So, it will be the best metric for the case of spam filtering.

The following plots are generated after applying AUCROC on my project both for training with 10-fold cross validation and testing the data without it:

