NaiveBayes_LogisticRegression_kNearestNeighbors

Samyuel Danyo, 02/2019

Introduction

In this report I present my implementations of four pattern recognition methods: Beta-Binomial Naive Bayes, Gaussian Naive Bayes, Logistic Regression with Newton's Method and L2 regularization learning, K-Nearest Neighbors with cross-validation, as well as, the results and analysis of the SPAM E-mail Dataset, classified with each of the above-mentioned algorithms.

The dataset is comprised of 4601 samples, each with 57 features. Out of them 1813(39.4%) are labeled as spam, leaving 2248(60.6%) as non-spam. The full description of the SPAM E-mail dataset and clarification on what is assumed as spam can be seen in [1].

The dataset is randomized and divided into two sub-sets – training with 3065 samples and a test (sometimes also used as validation) with 1536 samples. Two data-transformation techniques are applied to the training and test sets. A binarization is performed in order to prepare the dataset for the Beta-Binomial Naive Bayes fit. A log-transform is performed in order to prepare the dataset for the rest of the fits.

For classification analysis of the model fit, I observe a few different metrics: accuracy, overall error rate(1-accuracy), false positives error rate and false negatives error rate.

While the main metrics for evaluating the performance of the methods is the resultant accuracy/overall error rate, for the specific case of the SPAM E-mail dataset – a deeper insight can be drawn from the false positives/negatives error rate. As mentioned by [1], the false positives (classifying a non-spam email as spam) is very undesirable as this can lead to the loss of important correspondence. Hence, when I discuss the performance of the model fits, a special attention is given to the false positives rate with the aim of minimizing it.

Additionally, the effect that some hyperparameters have on the model fit are studied: for the Beta(***a***, ***a***)-Binomial Naive Bayes – the effect of the value of the "***a***" hyperparameter (hence of the prior); for the Logistic Regression method – the amplitude of the weight decay; for K-Nearest Neighbor – the value of "***k***".

The four methods are implemented in Python from scratch, using NumPy for matrices manipulation and calculations. Four shared helper functions are implemented for extracting the evaluation metrics and plotting a confusion table. Further method-specific helper functions are implemented for wrapped training, testing, displaying results and complete fitting using matplotlib for plotting graphs and some other basic helper packages.

For each method graphs of {train/test} {error rate, accuracy, false negatives error rate, false positives error rate} over {(hyperparameter) fit / training(for Logistic Regression)} are plotted. Based on the train/test predictions vs targets, confusion tables are displayed. Last but not least, the optimal value for the hyperparameter at question is chosen and the corresponding best model-fit performance is displayed.

Beta-Binomial Naive Bayes Classifier

Design

The classifier is designed to be configurable to either use the **M**aximum **L**ikelihood (**ML**) **E**stimation or to assume a prior Beta(***a***, ***a***) (**P**osterior **P**redictive (**PP**)) for each – the class priors probability and the class-conditional probabilities. Depending on its configuration, the classifier can use a pure **ML**, a Bayesian **PP** or a mixture for training and prediction.

Implementation

The Classifier Class

The classifier is implemented as a class with two fields {*C_Prior(2)*, *C_Cond(2, 2, n_features)*} for class priors and class-conditional probabilities respectively, and three methods:

- *__init__(n_features)* for declaring the fields and initializing them to '*0*'s.
- *train(X_train, T_train, beta_par_1, beta_par_2, beta_prior = False, beta_cond = False)* for training the classifier by calculating the log class-priors & class-conditional probabilities. The choice between **MLE** or **PP** is controlled by the *beta_{prior/cond}* inputs. For the class-priors - the method calculates the *P(y=1)* and sets the P(y=0) as *(1-P(y=1))*. For the class-conditional probabilities a similar thing is done – for each class *c={0,1}*, the method calculates the *P(X=1/c)* and then sets the *P(X=0/c) = (1-P(X=1/c)).*
- *predict(X_pred)* for making a prediction about the input-samples' labels, based on its model. The method calculates the probabilities for the samples to be of each class by summing the respective log class-priors and the class-conditional probabilities depending on the features' values and makes a prediction for the samples' labels based on which one is higher.

The Helper Functions

The Beta-Binomial Naive Bayes has two helper functions:

- *fit_beta_classifier(classifier, X_train, T_train, X_test, T_test)* for performing the classifier's training, prediction and data gathering for all Beta distributions.
- *show_results_beta(outputs…, T_train, T_test, beta_par, errors…, accuracies…, t_time)* for displaying all data gathered from the fit.

Experimental Setup

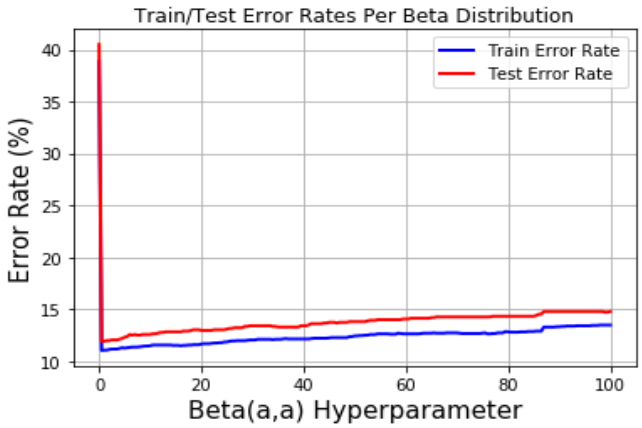
The fit is carried out on the binary dataset with **MLE** for the class priors and **PP** for the class-conditionals for each Beta hyperparameter $\boldsymbol{a} \in \{0, 0.5, 1, 1.5, 2, \cdot\cdot\cdot, 100\}$.

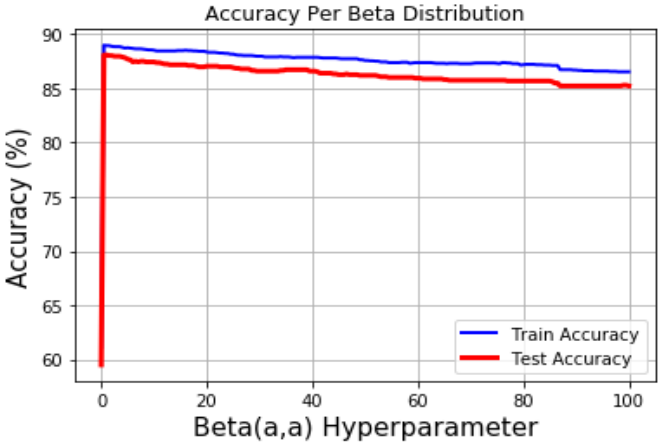
Results and Discussion

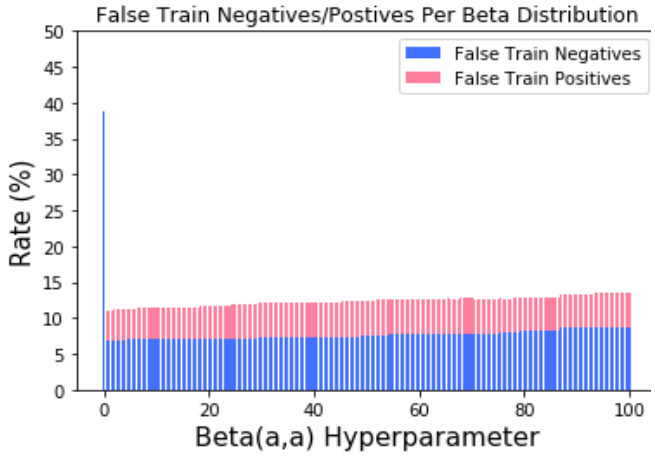
As seen on the plots of error rates versus the values of the Beta hyperparameter below, for *a=0*, the model performs very bad. This is to be expected as with *a=0*, the training assumes no prior, which makes the model prone to black-swan events and other disturbances of the fitted probabilistic space. Once a Beta prior is assumed, the error rate drops to a reasonable value of around *11%*. Another observation that can be made is the degradation of the model performance with increasing priors. This is due to the fact that if the prior is big, the significance of the training data is reduced and hence information is lost.

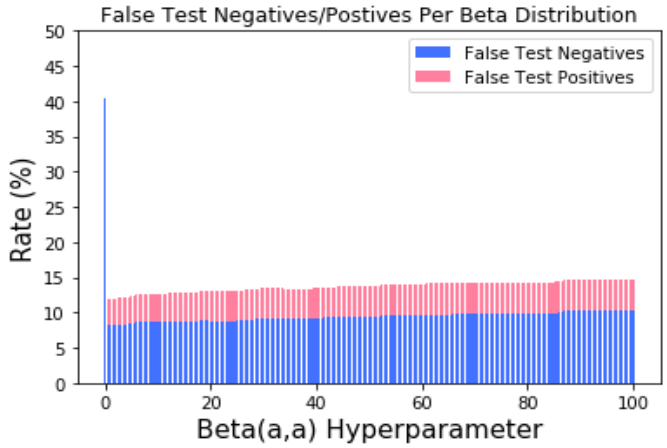
This results in a looser fit of the model, which puts increasing importance on the prior rather than the training data's features.

The breakdown of the error rate into false negatives/positives show that while the overall error rate is over *11%*, the false positives are a considerably small proportion of it (for the optimal beta – *3.65%*), which is very good for classifying spam and non-spam.



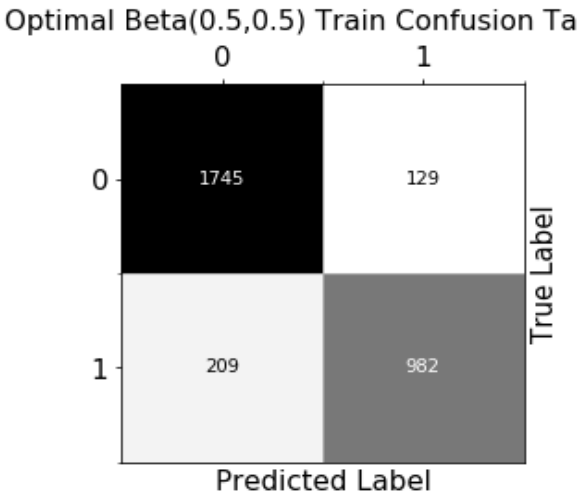


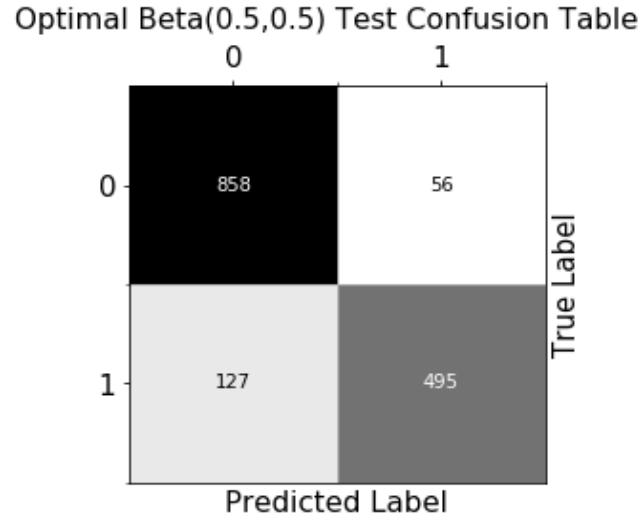




Results for *a* = {*1, 10, 100*} and the optimal *Beta(a, a)* fit.

```
BAYES CLASSIFIER(Beta(1.0,1.0))
TRAIN ACCURACY: 88.93964110929853% | TEST ACCURACY: 88.0859375%
TRAIN ERROR RATE: 11.06% | TEST ERROR RATE: 11.91%
TRAIN FALSE NEGATIVES ERROR RATE: 6.85% | TEST FALSE NEGATIVES ERROR RATE: 8.27%
TRAIN FALSE POSITIVES ERROR RATE: 4.21% | TEST FALSE POSITIVES ERROR RATE: 3.65%
--------------------------------------------------------------------------------
BAYES CLASSIFIER(Beta(10.0,10.0))
TRAIN ACCURACY: 88.51549755301794% | TEST ACCURACY: 87.43489583333334%
TRAIN ERROR RATE: 11.48% | TEST ERROR RATE: 12.57%
TRAIN FALSE NEGATIVES ERROR RATE: 7.05% | TEST FALSE NEGATIVES ERROR RATE: 8.66%
TRAIN FALSE POSITIVES ERROR RATE: 4.44% | TEST FALSE POSITIVES ERROR RATE: 3.91%
--------------------------------------------------------------------------------
BAYES CLASSIFIER(Beta(100.0,100.0))
TRAIN ACCURACY: 86.5252854812398% | TEST ACCURACY: 85.22135416666666%
TRAIN ERROR RATE: 13.47% | TEST ERROR RATE: 14.78%
TRAIN FALSE NEGATIVES ERROR RATE: 8.78% | TEST FALSE NEGATIVES ERROR RATE: 10.29%
TRAIN FALSE POSITIVES ERROR RATE: 4.70% | TEST FALSE POSITIVES ERROR RATE: 4.49%
--------------------------------------------------------------------------------
OVERALL TIME FOR CLASSIFIER FIT: 0.0h:0.0m:3.23319s
================================================================================
BEST PERFORMANCE ACHIEVED FOR: Beta(0.5,0.5)
TRAIN ACCURACY: 88.97226753670473% | TEST ACCURACY: 88.0859375%
TRAIN ERROR RATE: 11.03% | TEST ERROR RATE: 11.91%
TRAIN FALSE NEGATIVES ERROR RATE: 6.82% | TEST FALSE NEGATIVES ERROR RATE: 8.27%
TRAIN FALSE POSITIVES ERROR RATE: 4.21% | TEST FALSE POSITIVES ERROR RATE: 3.65%
================================================================================
```





Gaussian Naive Bayes Classifier

Design

For training the classifier uses the **M**aximum **L**ikelihood (**ML**) **E**stimation for the class-priors and the class-conditional mean and variance for each feature. The prediction is made by utilizing **MLE** as a plug-in estimator.

Implementation

The Classifier Class

The classifier is implemented as a class with three fields {*C_Prior(2)*, *C_Mean(2, n_features), C_Var(2, n_features)*} for class priors and class-conditional means and variances respectively; and four methods:

- *__init__(n_features)* for declaring the fields and initializing them to '*0*'s.
- *train(X_train, T_train)* for training the classifier by calculating the log class-priors & class-conditional means/variances. For the class-priors – using **MLE** the method calculates the *P(y=1)* and sets the P(y=0) as *(1-P(y=1))*. For the class-conditional probabilities – for each class *c={0,1}*, the method calculates the *means and variances for each feature.*
- *calculate_log_probability(X_pred) is the* plug-in estimator of the natural log of the class-conditional probabilities using the probability density function of the Gaussian (normal) distribution.
- *predict(X_pred)* for making a prediction about the input-samples' labels, based on the trained priors & the plug-in estimator's returned log class-conditional probabilities. The method calculates the probabilities for the samples to be of each class by summing the respective log class-priors and the class-conditional probabilities and makes a prediction for the samples' labels based on which one is higher.

The Helper Functions

The Gaussian Naive Bayes has two helper functions:

- *fit_gaussian_classifier(classifier, X_train, T_train, X_test, T_test)* for performing the classifier's training, prediction and data gathering.
- *show_results_gaussian(outputs…, T_train, T_test, errors…, accuracies…, t_time)* for displaying all data gathered from the fit.

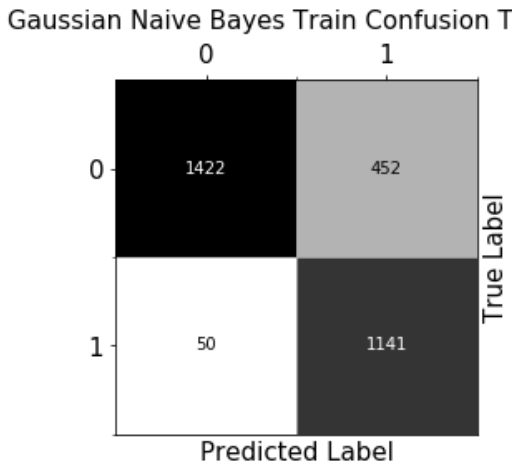
Experimental Setup

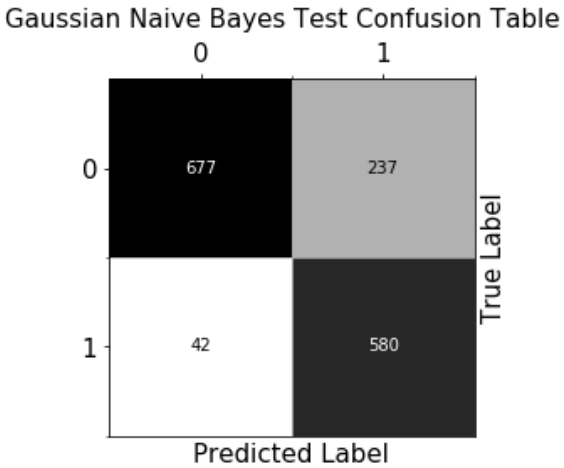
The fit is carried out on the log dataset.

Results and Discussion

As seen by the results below, the **MLE** Gaussian Naive Bayes classifier does not perform on the level of the **ML/PP** Beta-Binomial. The Gaussian classifier not only yields a significantly higher overall test error rate of *18.16%* (vs *11.91%)* but it also has predominantly false positives at *15.43%*, which is extremely high compared to the *3.65%* for the optimal Beta prior. This is probably due to the lack of assumed prior, which can cause distortions. The Gaussian Naive Bayes still performs much better than the Beta-Binomial Naive Bayes without assumed prior due to it being continuous and so capturing more information from the input samples, which translates to a better pattern modelling.

```
>>>>> GAUSSIAN NAIVE BAYES CLASSIFIER <<<<<
TRAIN ACCURACY: 83.6215334420881% | TEST ACCURACY: 81.8359375%
TRAIN ERROR RATE: 16.38% | TEST ERROR RATE: 18.16%
TRAIN FALSE NEGATIVES ERROR RATE: 1.63% | TEST FALSE NEGATIVES ERROR RATE: 2.73%
TRAIN FALSE POSITIVES ERROR RATE: 14.75% | TEST FALSE POSITIVES ERROR RATE:15.43%
OVERALL TIME FOR CLASSIFIER FIT: 0.0h:0.0m:0.05000s
```





Logistic Regression

Design

The Logistic Regression classifier is built as an artificial neuron from three layers: linear – containing the model parameters, which after a forward step yield the log-odds prediction; logistic – the sigmoid function which squashes the log-odds into [*0:1*] interval and yields the posterior probabilities; step – the step function with a threshold of *0.5*, which translates the logistic decision boundary at 0.5 to binary predictions. The neuron learns though the Newton's Method by doing a second order derivative cost estimation using both – the Hessian and the Gradient. The learning also utilizes a L2 regularization.

Implementation

The Classifier Class

The neuron is implemented as a class with two fields {weights*(n_features), bias(1)*} for model parameters; and five methods:

- *__init__(n_features)* for declaring the fields and initializing the bias to '*0*' and the weights to values from the "standard normal" distribution in between *[-1:1]*.

- *get_log_odds(X_train)* for performing the linear forward pass (*X.dot(w) + b*) which in the Logistic Regression case returns the log-odds estimation.
- *get_posterior(log_odds)* for calculating the *Sigmoid(log_odds)*, which in the Logistic Regression case is the posterior probabilities estimation.
- *get_output(posterior) for* performing the prediction for the samples' labels by applying *Step(posterior)*.
- *update_params_Newton(Posterior_train, T_train, X_train, L2_rate, regularization = False) for c*alculating the Newton's Method costs with L2 regularization & updating the neuron's parameters. The method appends '*1*'s for bias update to the input samples and '*0*'s to the weights vector for avoiding bias regularizing. After building the S, I matrices, the method calculates the cost updates for the parameters using the Hessian & Gradient.
- update_params_back_prop(self, Posterior_train, T_train, X_train, L2_rate, regularization = False) for calculating the Back Propagation Gradient Descent costs with L2 regularization & updating the neuron's parameters by given learning rate.

The Helper Functions

The Logistic Regression has six helper functions:

- *forward_step(LogisticNeuron, X_pred)* for computing and returning the posterior probabilities and output predictions. Utilizes get_loggs(), get_posterior() and get_output().
- *train_logistic(LogisticNeuron, X_train, T_train, X_val, T_val, L2_rate, NewtonUpdate)* for performing the forward propagation & update of the model parameters until convergence for a set L2 regularization rate. Implements Sequential/Minibatches/Batch learning, as well as, data gathering for the training. The method implements training through either Newton's Method or Back Prop. Utilizes forward_step() update_params_Newton(), update_params_back_prop().
- *show_results_logistic (X_train, X_val, nb_of_epochs, nb_of_batches, errors…, accuracies…, t_time)* for displaying all data gathered during training per epoch.
- *show_results_logistic_per_regularization (X_train, X_test, reg_rates, errors…, accuracies…, fit_time)* for displaying all data gathered from the fit per regularization rate.
- *test_logistic(LogisticNeuron, X_test, T_test) for* testing the learnt fit and gathering data about its performance.
- *fit_logistic(Regularization_rates, X_train, T_train, X_val, T_val, X_test, T_test)* - wrapper function which encapsulates training, testing, data gathering & data display for all regularization rates.

Experimental Setup

The fit is carried out on the log dataset in two configurations:

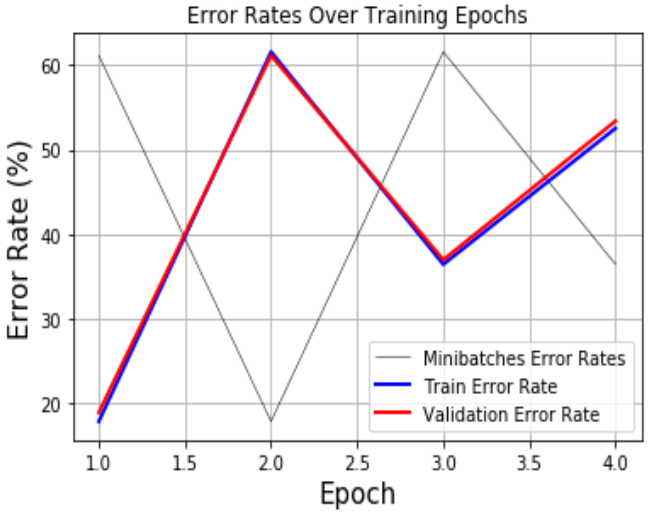
- Newton's Method - batch learning for regularization rates: ***L2_rate*** ∈ {1, 2, · · ·, 9, 10, 15, 20, · · ·, 95, 100}.
- Backpropagation Gradient Descent – sequential learning with *learning_rate=0.002* for regularization rates: ***L2_rate*** ∈ {1, 2, · · ·, 9, 10, 15, 20, · · ·, 95, 100}/1000.

Results and Discussion

Since Newton's Method learning makes a second order estimations, the model converges very fast - in five epochs on average. As can be seen from the error-rate/accuracy vs regularization rate plots bellow, while for some rates the model fits extremely well with error rates under *7%*, for others the errors rates are high. If a closer look is taken, it becomes apparent that during some of the fits, the Hessian gets badly conditioned or singular. This is

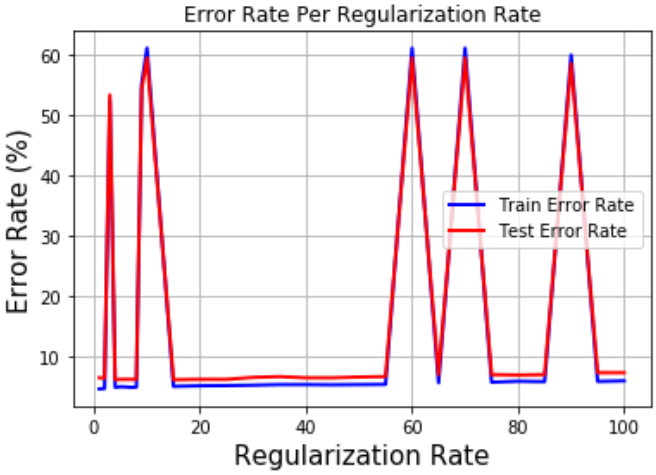
due to the fast convergence and big absolute values of regularization rate. Since the classifier implements L2 regularization, if a weight has become big in its first update, during its second one the penalty would be so big that the weight could become extremely small. This kind of oscillations, caused by the high velocity of the descent towards the global minimum could be the reason for the observed drastic variations in accuracy and error rate. The mentioned oscillations can be observed in the plot “Error Rates Over Training Epochs” bellow, which shows the train, validation and batch error rates during training in one of the drastic L2_rate cases.

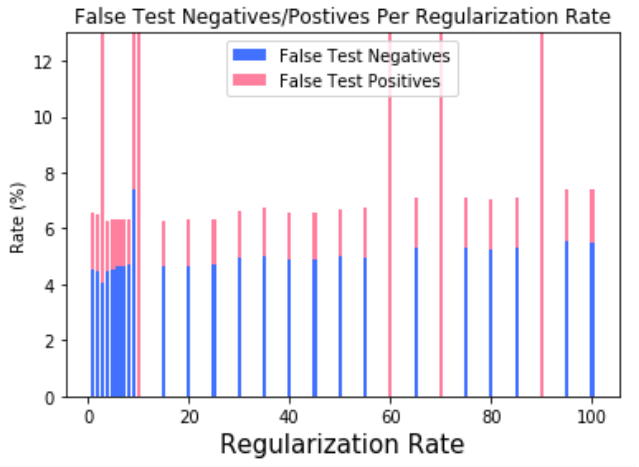
When, looking at the well-converged models in the error rate/accuracy plots over L2_rate below, it can be seen that lower regularization rates provide better performance. With the optimal L2_rate, the Logistic Regression achieves a test error rate of *6.25%* and a false positives rate of only *1.76%* drastically outperforming the Naive Bayes!



A Backpropagation Gradient Descent fit was ran, too. As can be seen by the plots below, the drastic oscillations are not present, which supports the above-mentioned deductions. The resultant data clearly shows smaller L2_rates are favorable with the optimal L2_rate=0.009, test error rate = 5.92% and false positives rate of 1.76%!

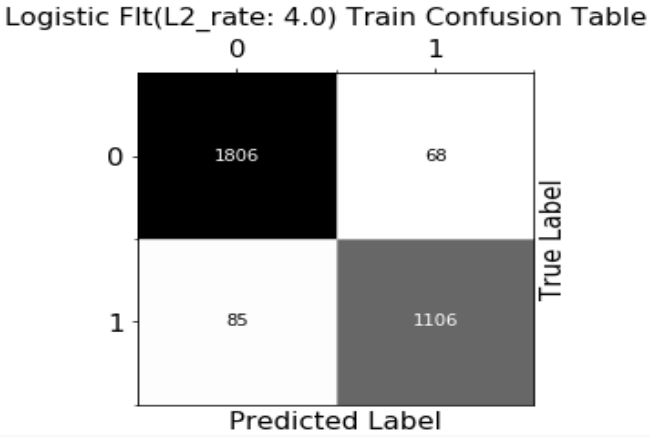
Newton’s Method Results:

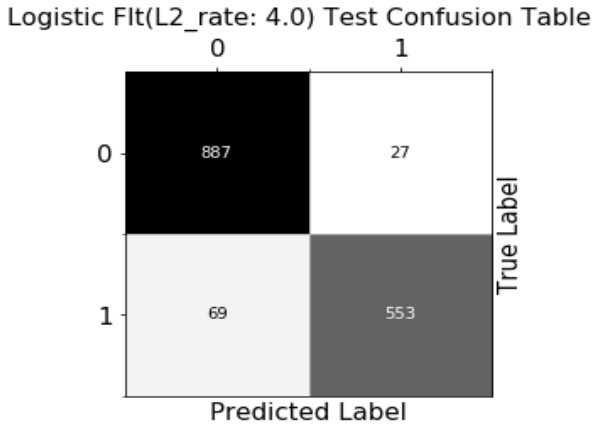




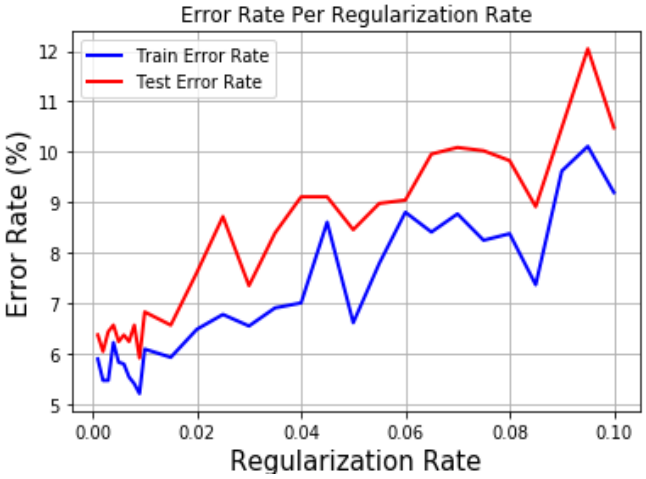

```
-------------------------------------------------------------------------------
>>>>> LOGISTIC REGRESSION NEURON(L2_rate = 1.0) <<<<<
TRAIN ACCURACY: 95.27% | TEST ACCURACY: 93.42%
TRAIN ERROR RATE: 4.73% | TEST ERROR RATE: 6.58%
TRAIN FALSE NEGATIVES ERROR RATE: 2.58% | TEST FALSE NEGATIVES ERROR RATE: 4.56%
TRAIN FALSE POSITIVES ERROR RATE: 2.15% | TEST FALSE POSITIVES ERROR RATE: 2.02%
-------------------------------------------------------------------------------
>>>>> LOGISTIC REGRESSION NEURON(L2_rate = 10.0) <<<<<
TRAIN ACCURACY: 38.86% | TEST ACCURACY: 40.49%
TRAIN ERROR RATE: 61.14% | TEST ERROR RATE: 59.51%
TRAIN FALSE NEGATIVES ERROR RATE: 0.00% | TEST FALSE NEGATIVES ERROR RATE: 0.00%
TRAIN FALSE POSITIVES ERROR RATE: 61.14% | TEST FALSE POSITIVES ERROR RATE:59.51%
-------------------------------------------------------------------------------
>>>>> LOGISTIC REGRESSION NEURON(L2_rate = 100.0) <<<<<
TRAIN ACCURACY: 93.93% | TEST ACCURACY: 92.58%
```

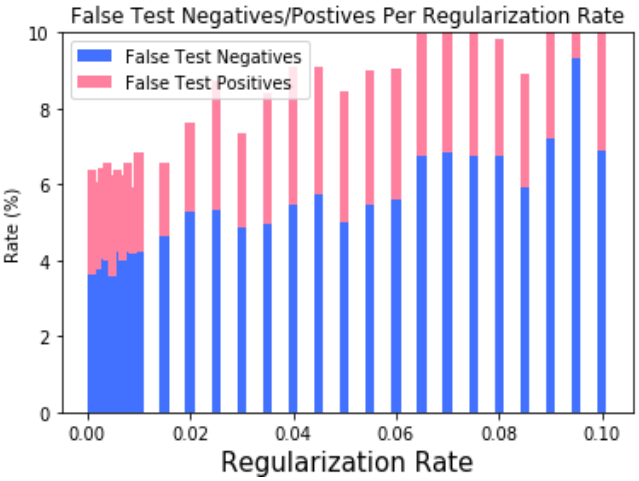
```
TRAIN ERROR RATE: 6.07% | TEST ERROR RATE: 7.42%
TRAIN FALSE NEGATIVES ERROR RATE: 3.88% | TEST FALSE NEGATIVES ERROR RATE: 5.47%
TRAIN FALSE POSITIVES ERROR RATE: 2.19% | TEST FALSE POSITIVES ERROR RATE: 1.95%
--------------------------------------------------------------------------------
OVERALL TIME FOR FIT: 0.0h:0.0m:38.08818s
================================================================================
BEST PERFORMANCE ACHIEVED FOR: L2_rate = 4.0
TRAIN ACCURACY: 95.00815660685154% | TEST ACCURACY: 93.75%
TRAIN ERROR RATE: 4.99% | TEST ERROR RATE: 6.25%
TRAIN FALSE NEGATIVES ERROR RATE: 2.77% | TEST FALSE NEGATIVES ERROR RATE: 4.49%
TRAIN FALSE POSITIVES ERROR RATE: 2.22% | TEST FALSE POSITIVES ERROR RATE: 1.76%
================================================================================
```





Backpropagation Gradient Descent Results:






```
--------------------------------------------------------------------------------
>>>>> LOGISTIC REGRESSION NEURON(L2_rate = 0.001) <<<<<
TRAIN ACCURACY: 94.09% | TEST ACCURACY: 93.62%
TRAIN ERROR RATE: 5.91% | TEST ERROR RATE: 6.38%
TRAIN FALSE NEGATIVES ERROR RATE: 2.64% | TEST FALSE NEGATIVES ERROR RATE: 3.65%
TRAIN FALSE POSITIVES ERROR RATE: 3.26% | TEST FALSE POSITIVES ERROR RATE: 2.73%
--------------------------------------------------------------------------------
>>>>> LOGISTIC REGRESSION NEURON(L2_rate = 0.01) <<<<<
TRAIN ACCURACY: 93.90% | TEST ACCURACY: 93.16%
TRAIN ERROR RATE: 6.10% | TEST ERROR RATE: 6.84%
TRAIN FALSE NEGATIVES ERROR RATE: 3.16% | TEST FALSE NEGATIVES ERROR RATE: 4.23%
TRAIN FALSE POSITIVES ERROR RATE: 2.94% | TEST FALSE POSITIVES ERROR RATE: 2.60%
--------------------------------------------------------------------------------
>>>>> LOGISTIC REGRESSION NEURON(L2_rate = 0.1) <<<<<
TRAIN ACCURACY: 90.80% | TEST ACCURACY: 89.52%
TRAIN ERROR RATE: 9.20% | TEST ERROR RATE: 10.48%
TRAIN FALSE NEGATIVES ERROR RATE: 5.25% | TEST FALSE NEGATIVES ERROR RATE: 6.90%
TRAIN FALSE POSITIVES ERROR RATE: 3.95% | TEST FALSE POSITIVES ERROR RATE: 3.58%
```

```
--------------------------------------------------------------------------------
OVERALL TIME FOR FIT: 0.0h:1.0m:21.36165s
================================================================================
BEST PERFORMANCE ACHIEVED FOR: L2_rate = 0.009
TRAIN ACCURACY: 94.77977161500816% | TEST ACCURACY: 94.07552083333334%
TRAIN ERROR RATE: 5.22% | TEST ERROR RATE: 5.92%
TRAIN FALSE NEGATIVES ERROR RATE: 3.03% | TEST FALSE NEGATIVES ERROR RATE: 4.17%
TRAIN FALSE POSITIVES ERROR RATE: 2.19% | TEST FALSE POSITIVES ERROR RATE: 1.76%
================================================================================
```

K-Nearest Neighbors

Design

The classifier measures the “nearness” between train and test samples via the Euclidean distance. The optimal *K* is chosen via a five-fold cross-validation.

Implementation

The Classifier Class

The classifier is implemented as a class with two fields {X_train, T_train} for the training dataset, and four methods:

- *__init__()* for declaring the classifier fields.
- *train(X_train, T_train)* for Initialize the classifier fields with the training dataset.
- *euclidean_distance(X_pred) for* calculating the Euclidean distances between the input samples (X_pred) and the training samples.
- *predict(X_pred, k)* for making a prediction about the input-samples' labels by calculating the Euclidean distances between them & the training samples, and picking the most common label in the k-nearest training samples.

The Helper Functions

The K-Nearest Neighbors has four helper functions:

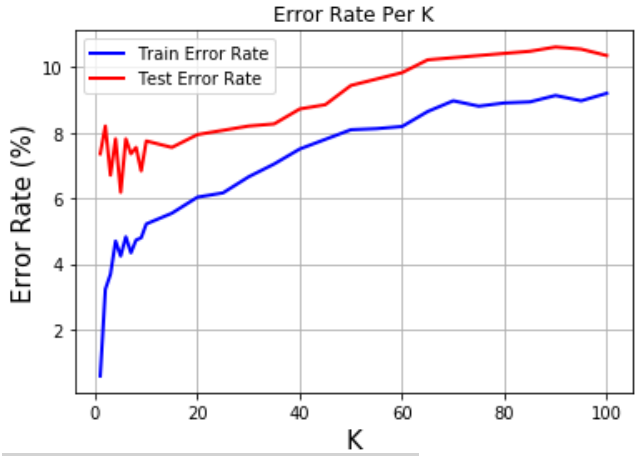
- *cross_validate(KNearestNeighbors, X_train, T_train, K_values)* for performing a five-fold cross-validation using the train dataset to pick the optimal K in between the K_values.
- *test_KNN(KNearestNeighbors, K, X_test, T_test)* for testing the learnt fit and gathering data about its performance.
- *fit_KNN(K_values, X_train, T_train, X_test, T_test)* for performing cross-validation, classifier training, testing and data gathering for all K_values.
- *show_results_KNN(X_train, X_test, K_values, cross-validation accuracy, errors…, accuracies…, fit_time)* for displaying all data gathered from the fit.

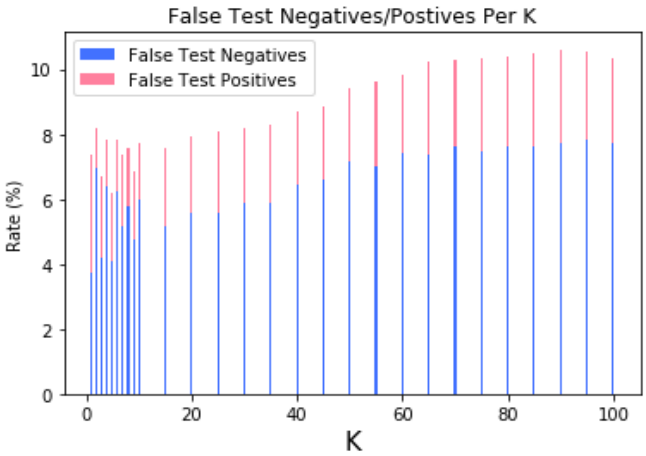
Experimental Setup

- The fit is carried out on the log dataset for each **k** ∈ {1, 2, · · ·, 9, 10, 15, 20, · · ·, 95, 100}.

Results and Discussion

As seen in the plots and data listings below, the optimal K is equal to 5 with error rate of 6.08% and false positives rate of 2.08%. When testing on the training set logically the best model-fit is for k=1 as the samples would just follow on top of each other and decreases with rising k, as the clusters intertwine. The accurate way to judge the model performance is by the cross-validation accuracy/testing accuracy. For k 1-4 the error rate is slightly higher than 6.5% as such a small number of samples would require extremely closely-clustered data. For k over 5 the error rate steadily grows as the clusters get too intertwined.






```
>>>>>>> kNN(K = 1) <<<<<<<
TRAIN ACCURACY: 99.41% | TEST ACCURACY: 92.64%
TRAIN ERROR RATE: 0.59% | TEST ERROR RATE: 7.36%
TRAIN FALSE NEGATIVES ERROR RATE: 0.13% | TEST FALSE NEGATIVES ERROR RATE: 3.71%
TRAIN FALSE POSITIVES ERROR RATE: 0.46% | TEST FALSE POSITIVES ERROR RATE: 3.65%
>>>>>>> kNN(K = 10) <<<<<<<
TRAIN ACCURACY: 94.78% | TEST ACCURACY: 92.25%
TRAIN ERROR RATE: 5.22% | TEST ERROR RATE: 7.75%
TRAIN FALSE NEGATIVES ERROR RATE: 3.39% | TEST FALSE NEGATIVES ERROR RATE: 5.99%
TRAIN FALSE POSITIVES ERROR RATE: 1.83% | TEST FALSE POSITIVES ERROR RATE: 1.76%
>>>>>>> kNN(K = 100) <<<<<<<
TRAIN ACCURACY: 90.80% | TEST ACCURACY: 89.65%
TRAIN ERROR RATE: 9.20% | TEST ERROR RATE: 10.35%
TRAIN FALSE NEGATIVES ERROR RATE: 6.33% | TEST FALSE NEGATIVES ERROR RATE: 7.75%
TRAIN FALSE POSITIVES ERROR RATE: 2.87% | TEST FALSE POSITIVES ERROR RATE: 2.60%
BEST PERFORMANCE ACHIEVED FOR: K = 5
TRAIN ACCURACY: 95.75856443719412% | TEST ACCURACY: 93.81510416666666%
TRAIN ERROR RATE: 4.24% | TEST ERROR RATE: 6.18%
TRAIN FALSE NEGATIVES ERROR RATE: 2.61% | TEST FALSE NEGATIVES ERROR RATE: 4.10%
TRAIN FALSE POSITIVES ERROR RATE: 1.63% | TEST FALSE POSITIVES ERROR RATE: 2.08%
```

Final Discussion

From the results, presented above, it can be deducted that both Logistic Regression and K-Nearest Neighbors approaches to classifying spam from non-spam E-mails are very good with overall error rates of around *6%* and false positives rates of around *2%*. While the Beta-Binomial Naive Bayes underperforms, it still yields good results of around *11.9%* error rate and *3.65%* false positives rate. It can prove useful in situations where extremely light solution is needed. The Gaussian Naive Bayes using **MLE** is the worst performer with comparatively very high false positives rate of over 15%**.** Its performance might be boosted by assuming a prior.

All in all, my experiment supports findings in [2] that for classifying spam emails the bigger weight is on the selection of training dataset, while the classification method is less important, since a number of simple algorithms can perform very well.

[1] *Web.stanford.edu*, 2019. [Online]. Available: https://web.stanford.edu/~hastie/ElemStatLearn/datasets/spam.info.txt. [Accessed: 22- Feb- 2019].

[2] S. Nizamani, N. Memon, M. Glasdam and D. Nguyen, "Detection of fraudulent emails by employing advanced feature abundance", *Egyptian Informatics Journal*, vol. 15, no. 3, pp. 169-174, 2014. Available: 10.1016/j.eij.2014.07.002.