ANALYSIS OF SENTIMENTAL LABELLED SENTENCES

COSC 522 - Machine Learning - Final Project

Authors

Kevin De Angeli
Hector D. Ortiz-Melendez

The University of Tennessee
Knoxville

December 11, 2019

Abstract

In this project, we apply multiple machine learning (ML) algorithms for the natural language processing task known as sentiment analysis. The data used contains people's reviews from Amazon, IMDb, and Yelp. Different parametric and non-parametric models have been implemented with two different feature extraction techniques. We compare the performance of our algorithms with the accuracy results provided by the team who made the dataset publicly available. We report accuracy, confusion matrices, and ROC curves for each model, and we extensively explore the hyper-parameter space of the models, when applicable. We show that some of our algorithms have performed better than their ML algorithms, but their deep learning (DL) models still provide the greater accuracy.

1 Introduction

Every day, millions of people post online their opinions about products, movies, places they visited, among many other activities and experiences. Together, this large amount of posts can be made into a dataset and fed to different machine learning and deep learning algorithms to understand people's general sentiment. In the context of e-commerce, customers' reviews can be useful for manufacturers because they tell what customer liked or disliked about a product [1]. However, you can now find hundreds or thousands of reviews for a single product, and inferring the general opinion of a large pool of comments is expensive and time consuming. Sentiment analysis is one of the tasks of natural language processing (NLP), a subfield of linguistics and computer science.

Scientist have approached sentiment analysis at different levels. Turney [2] developed an unsupervised learning algorithm to classify reviews as positive or negative. His work focuses on text analysis as a whole. Other authors have performed sentiment analysis at the sentence level. That includes Hu and Liu [1] who presented an algorithm to mine and summarize customer reviews of a product. However, unlike traditional text classification as a whole, their work focuses on identifying the features of the product on which the opinions are positive or negative. Recently, scientist have also been successful at performing sentiment analysis at the phrase level. For example Wilson et al. [3] presented a new approach for analysis at the phrase level to identify between neutral or polar expressions. Nevertheless, there are many more linguistic challenges to tackle such as ambiguous comments, specifically sarcasm identification where the sentiment is implied. Sarcasm is a challenging problem given its topi-dependency and highly contextual nature. Expressions like sarcasm can be approached with techniques such as user profiling [4].

Another important challenges of NLP is feature space. Most of the earlier work in NLP was developed using an unigram (1-gram), bag-of-words (BoW) approach. BoW assumes that text is just a set of words where order does not matter; the corpus of text is then represented as a "document-term matrix of counts" [5]. This basic assumption about text has proven to be successful in NLP, and it has been applied to topic modeling and other tasks such as reporting partisan terms from political speeches [5]. Another popular feature extraction technique is Term Frequency-Inverse Document Frequency (TF-IDF). TF-IDF basically tries to quantify how important a certain words is by observing how many times it appears in a specific document in comparison with how many times the word appears in the corpus [6]; words that appear frequently in a small number of documents but do not occur often in the whole corpus would have a high TF-IDF value. However, the apparent problem with unigrams is that they do not conserve the type of information that is inferred from the conjunction of two or more words. For example, phrases such as "social security" would completely lose their meaning under the BoW paradigm. One potential solution to this problem is $N-grams$. An N-gram refers to a sequence of ordered words where the length of the sequence is N. An $N-gram$ corpus provides information about how frequent a series of words occur [7]. Note that capturing the $N-grams$ of a corpus with dictionary size M will result in $M^N-grams$ [8]; in practice this number is prohibitively expensive, and we usually filter $N-grams$ which frequency are below certain threshold [7]. Another alternatives to the BoW model is parts-of-speech (POS). POS refers to mapping each word in a corpus to a specific part of speech (noun, pronoun, verbs, etc). This mapping process is known as part-of-speech tagging and is usually done with unsupervised learning algorithms that look into the contexts in which words occur to assign a specific label. POS are useful because they can provide a lot of information about a word and its neighbors [9]. The state of the art feature extraction technique is known as word embeddings. The idea behind word embedding is to "represent words as dense vectors that are derived by various training methods inspired from neural-network language modeling" [10]. One of the advantages of word embedding is that they attempt to capture word similarities. However, one significant drawback is that words can have multiple meanings, and that does not allow for some words to have a well-defined, single representation. One of the most popular word embedding algorithms is called Word2vec, and it was developed by a team lead by Tomas Mikolov at Google. Recently, Facebook has released a faster version of Word2vec called FastText. Since then, FastText has been used for numerous sentiment analysis tasks [11].

Scientists have developed sentiment analysis model for all kind of applications. Nogueira dos Santos et al. [12] have analyzed Twitter posts. This a challenging task because messages don't provide much information about the context. They used a deep convolutional network (traditionally implemented for image processing tasks) and obtaiened an accuracy of 85.7%. Wöllmer et al. [13] analyzed the general sentiment of online videos by considering not only textual information but also audio features. Thet et al. [14] proposed a method of automatic sentiment analysis of movies reviews that provides orientations (positive or negative) and different strength of these orientations. Gräbnera et al. [15] implemented a lexicon-based approach to classify hotel reviews. They obtained their data from the TripAdvisor website. Finally, aggression identification is an extreme application of sentiment analysis that has been enabled by cyber-harassment and cyber-bullying in social media [16].

The dataset used for our paper contains sentences labeled with positive and negative sentiment. The data was originally collected by Kotzias et al. [17]. In their work, they used the dataset to test their new algorithm (GICF). Their results are presented in Table 1. They have only reported the accuracy of three algorithms, and two of these are logistic regression with two different feature spaces. In this project several ML algorithms were trained: Gaussian classifiers, k-nearest neighbors (KNN), backpropagation neural network (BPNN), random forest (RF), classifier fusion, and support vector machine (SVM).

Model	Amazon	IMDb	Yelp
Logistic regression with BoW	79.086.3%	76.286.3%	75.186.3%
Logistic regression with Word Embeddings	54.386.3%	57.986.3%	66.586.3%
GICF with Word Embeddings	88.286.3%	86.086.3%	86.3%

Table 1: Kotzias et al. [17] results for the Amazon, Yelp, and IMDb reviews dataset. They called they novel model "GICF".

2 Methods

2.1 Dataset

The dataset selected for this project consists of three individual datasets containing people's reviews of movies, products, and restaurants from IMDb, Amazon, and Yelp, respectively. Each of these individual datasets consists of 1000 entries, where 500 are positive reviews, and 500 are negative reviews.

We have worked with these datasets individually, training the algorithms using one of these three datasets at the time. However, a fourth dataset was created by combining the other three datasets; we call this dataset "merged". Training the algorithms with the merged dataset can lead to interesting outcomes because the type of vocabulary used to review a movie, for example, can be significantly different than the type of words you may find on a product review.

2.1.1 Feature Extraction, Data Normalization, and Dimensionality Reduction

Two common feature extraction techniques in Natural Language Processing (NLP) were used: bag-of-words (BoW) and Term Frequency-Inverse Document Frequency (TF-IDF). Table 9 shows the dimension of the training datasets after obtaining the features. The number of columns represent the size of the dictionary of each of the datasets minus words such as articles that do not carry much information. Data normalization and dimensionality reduction was done in conjunction with BoW and TF-IDF. The dimension is reduced by removing "stop words". Stop words are words that do not carry much meaning, for example: "the", "an", "in", and "at". Because the BoW and TF-IDF approach considers each distinct words as a feature in the dataset, by removing stop words we end up with a reduced dataset.

Method	Amazon	IMDb	Yelp	Merged
BoW & TF-IDF	[1000 x 1848]	[1000 x 3047]	[1000 x 2035]	[3000, 5156]

Table 2: Dataset dimensions after features were extracted.

2.1.2 N-fold Cross Validation

N-fold cross validation is a validation technique in which the dataset is split into n subgroups. Then the ML/DL algorithm is trained with $n-1$ of these subgroups, and it's evaluated using the remaining subgroup of data. This process is repeated n times, resulting in n different accuracy scores. Then the average of the n accuracy results is computed. In this project, unless otherwise specified, all algorithms have been run with 10-fold cross validation.

2.2 Gaussian Classifiers

2.2.1 One-modal Gaussian: Case I

In Case I of the Gaussian classifiers, it is assumed that the standard deviation of the classes is the same and there is no correlation between the classes. This assumption implies that the features are statistically independent[18]:

$$\Sigma_i = \sigma^2 \mathbf{I}$$

Geometrically, Case I assumption corresponds to the case in which "the samples fall in equal-size hyperspherical clusters" [18] It is also important to notice that Case I classify data points based on the Euclidean distance between each point and the center of the clusters.

Under this assumption, the discriminant function takes the form:

$$g_i(x) = -\frac{1}{2\sigma^2}[\mathbf{x}^t\mathbf{x} - 2\mu_i^t\mathbf{x} + \mu_i^t\mu_i] + \ln P(\omega_i) \tag{1}$$

Even though Equation 1 seems to take the form of a quadratic equation, the term $\mathbf{x}^t\mathbf{x}$ is the same for all classes, making this equation a linear discriminant function [18]. Given that the dataset used consists of two classes, the decision boundary is simply calculated by solving:

$$g_1(x) = g_2(x)$$

This idea is applied to find the decision boundaries of Case I, II, and III. However, Richard O. Duda [18] points out an alternative way to calculate the decision boundary in case I:
Give the vector

$$w = \mu_i - \mu_j$$

and the point

$$\mathbf{x}_0 = \frac{1}{2}(\mu_i + \mu_j) - \frac{\sigma^2}{||\mu_i - \mu_j||^2} \ln \frac{P(\omega_i)}{P(\omega_j)}(\mu_i - \mu_j) \tag{2}$$

The decision boundary will also be given by the line which passes through $\mathbf{x}_0$ orthogonal to the vector w. Additionally, note that in Equation 2, the right part containing ln becomes 0 when the prior probabilities are equal. Therefore, these identities provide a simple and quick way to compute the decision boundary.

2.2.2 One-modal Gaussian: Case II

The second case of the Gaussian classifiers assumes that the covariance matrices of both classes are the same, that is

$$\Sigma_i = \Sigma$$

Geometrically, this assumption "corresponds to the situation in which the sample fall in hyperllipsoidal clusters of equal size and shape" [18]. In contrast to Case I, Case II classifies data points based on Mahalanobis distance which considers not only the distance between the data point and the center of the cluster, but also takes into consideration the covariance matrix of the classes. Note that just like in Case I, the decision boundary of Case II is also defined as a linear function.
A natural question to ask is how to choose the common covariance matrix. Assuming no correlation between classes, there are two methods that seem to justify the two entries of the matrix intuitively:

1. Use the standard deviation of the first and second columns of the data set as the two entries.
2. Use the average of the two standard deviations of the two columns when $y = 0$ and the average of the two standard deviations when $y = 1$

Under the assumptions of Case II, the discriminant functions can be simplified as:

$$g_i(x) = (\Sigma^{-1}\mu_i)^t\mathbf{x} - \frac{1}{2}\mu_i^t\Sigma^{-1}\mu_i + \ln P(\omega_i) \tag{3}$$

2.2.3 One-modal Gaussian: Case III

The third Baysian classifier makes not assumption about the covariance matrix. Each discriminant function has their own covariance matrix calculated based on the statistics of the data set. Here, the discriminant function can not be simplified and take the following form:

$$g_i(x) = [\mathbf{x}^t(-\frac{1}{2}\Sigma_i^{-1})\mathbf{x}] + [(\Sigma_i^{-1}\mu_i)^t\mathbf{x}] - \frac{1}{2}[\mu_i^t\Sigma_i^{-1}\mu_i + \ln|\Sigma_i|] + \ln P(\omega_i) \tag{4}$$

2.3 kNN

The idea behind kNN is very simple: given a test sample, find the k points in the dataset that are the closest to the test point. We call these points the "neighbors" of the test point. Then, classify the test point based on the discriminant: $\frac{k_i}{k}$ where k_i represents the number of data point from class i in the neighborhood of size k.

kNN can also be understood in terms of posterior probabilities:

$$p(w_i|x) = \frac{p(x|w_i)p(w_i)}{p(x)}$$

$$p(w_i|x) = \frac{\frac{k_i}{n_iV}\frac{n_i}{n}}{\frac{k}{nV}} = \frac{k_i}{k}$$

In terms of algorithms, I have written 3 functions that work together to classify data based on the k closest Neighbors. The function KNN accepts the training and testing data set, and an integer value for k. This function loops through the testing data set and calculates the accuracy. The function $euclidiandsitance$ computes the distance between the test point that it receives and every other point in the training data set. Then, it sends the results to $guessLabel$, which sorts the array of data and makes a guess based on the k closest training data points.

2.4 Perceptron and BPNN

Perceptron is a type of artificial neuron. They were created by Frank Rosenblatt between the 1950s and 1960s, and they were inspired by Warren McCulloch and Walter Pitts' earlier work [19]. Perceptrons take multiple inputs and output a single binary element. The perceptron contains weights that are real numbers being multiplied by each of the input values. The weights represent the relative importance of each of the input parameters [19]. The perceptron decides the output value based on the simple rule (5):

$$Output = \begin{cases} 0 & if \sum_j dot(w,x) + b \leq 0 \\ 1 & if \sum_j dot(w,x) + b > 0 \end{cases} \tag{5}$$

where w are the weights associated with each of the x input parameters. In this model b can be thought as a threshold that established how easy it's for the model to output a 1. So, for a large b, the model will have certain tendency to output 1 unless evidence shows that the output should be 0.

In the context of nueral networks, "learning" refers to finding the appropriate $w's$ and $b's$ that will minimize the error (difference between the model output and the actual, expected output). For this task, one would naturally want to modify the weights so that small changes in one weight would produce small changes in the output [19]. However, the perceptron model does not follow this rule. In other words, small changes in the weight of the perceptron model can lead to completely opposite outputs. In other words, there is not a simple rule to update the weights correctly. The Sigmoid neuron model was implemented to solve this specific problem. That is, Sigmoid neurons allow for small changes in the weights to lead to small changes in the output. Sigmoid neurons work just like perceptrons, the only difference is that the output is not 0 and 1; instead, the output is defined based on the Sigmoid function:

$$Output = \frac{1}{1+e^{-[dot(w,x)+b]}}$$

Note that the output of the sigmoid function is similar to that of the perceptron since whenever $dot(w,x)+b$ is large, the sigmoid function will return a value close to 1; otherwise, it will return a value close to 0. Another main takeaway from the sigmoid function is that its shape is smooth, and we know from calculus (partial derivatives) that this implies that small changes in w and b leads to small changes in the output.

The basic neural network architecture is just a group of sigmoid neurons that contain a greater number of weights and allow for a more complex decision boundary. Figure 1 presents one possible architecture. Note that the number of hidden layers and the number of neurons in each layers are arbitrary parameters. In fact, one of the main purposes of this paper is to quantify how changes in these parameters affect the overall accuracy.

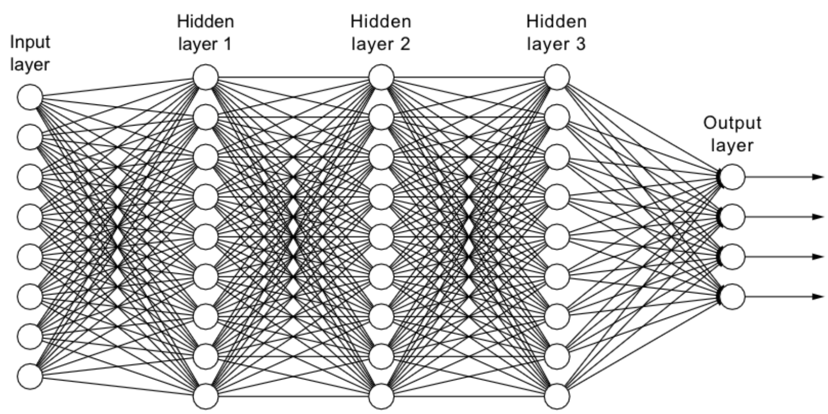


Figure 1: Example architecture of a Neural Network.

2.4.1 Gradient Descent and Stochastic Gradient Descent

The goal of the neural network presented in this paper is to predict certain label y given some input vector $\vec{x}$. In order to predict the output correctly most of the time, we need to be able to quantify the error so we can adjust the weights of the neural network to minimize some cost function. Equation (6) presents the mean square error (MSE), which serves as our cost function. Here, n is the number of training data points, the $x's$ are the training data point, $y(x)$ is the actual label of the associated data points, and w and b are the weights and biases of the network.

$$C(w,b) = \frac{1}{2n}\sum_x ||y(x)-\alpha||^2 \tag{6}$$

Gradient descent is of many optimization algorithm that we can use to find the minimum of a cost function. Because of the extensive number of parameters present in the cost function of a neural network, analytical techniques are not feasible. We need some type of iterative algorithm. Gradient descent is based on the idea that for some arbitrary region of a differentiable function, one will decrease faster if moving in the direction of the negative gradient of that function. This idea is represented mathematically in Equation 7.

$$v \rightarrow v' = v - \eta \nabla C \tag{7}$$

This equation states that if you find yourself in some position v, you will head to a lower position if you move in the direction of $-\nabla C$ with some step of size η. Note that the gradient vector ∇C is defined as the vector containing the partial derivative of each variable of the cost function (Equation 8).

$$\nabla C = (\frac{\partial C}{\partial v_1}, ..., \frac{\partial C}{\partial v_n}) \tag{8}$$

We usually choose the value of η to be small, so that $\nabla C < 0$, but we also do not want η to be extremely small because that will produce the algorithm to move too slowly [19] . There is not a rule to pick η and some experimentation is required. An example analysis of the impact of different η is presented later in this paper.

Note that based on Equation 7, to compute the gradient ∇C, we need to calculate $||y(x) - \alpha||^2$ for each data point. This could take long, and it will make the learning process slow. To solve this problem, we can use Stochastic Gradient Descent, the idea if that we can update the weights by computing $||y(x) - \alpha||^2$ of a small, random selection of training samples, which would consequently speed up the learning process [19].

2.5 SVM

Support Vector Machine uses the extreme data points (points that are close to the opposite classes) to find margins and hyper-planes which in turn serve as decision boundaries. However, not all datasets are linearly separable. For this, SVM uses Kernel functions to transform non-linear spaces into linear spaces, so that an hyper-plane is possible to find. Some examples of popular kernel types are Polynomial Kernel, Radial Basis Function RBF kernel, and sigmoid kernel. Figure 2 shows an example of applying SVM to a simle 2D dataset.

2.6 Random Forest

A random forest consists of multiple decision trees packed together. Decision trees is a tool to make decision using a tree-like model which consists of nodes and leaves that connect the nodes. At a high level, the algorithm can be interpreted as a chain of “if statements" [20]. In a decision tree, internal nodes represent some type of "test" on an attribute of the dataset, branches represent the outcome of the test, and leaf nodes represent class labels [20]. Decision trees can be easy to interpret and they could lead to high accuracy results even on non-linear problems.

2.7 K-means

K-means is a clustering algorithm generally used for unsupervised learning tasks. However, we implemented it in this project by setting the number of clusters equal to the number of classes (2).

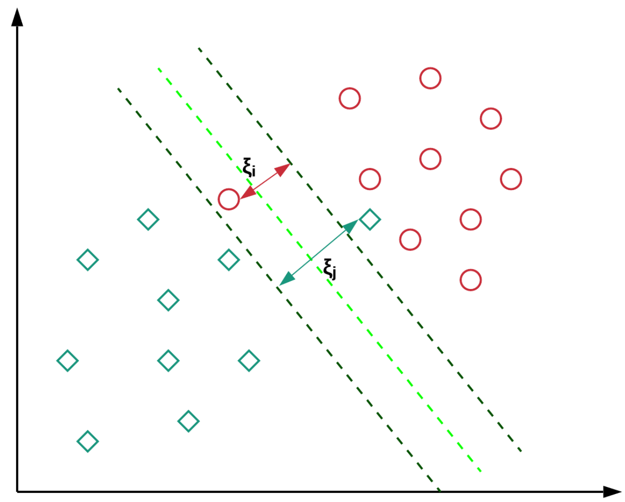


Figure 2: Example application of SVM to a simple 2D dataset.

After that the clusters converge to a definite position, the cluster class is decided based on majority vote of the points within each cluster. Then, for any test point, we used euclidean distance to find the closest cluster and classify the test point accordingly.

The steps of the algorithms is as follows:

1. Select the number of clusters K based on the number of classes in the training dataset.
2. Assign the coordinate of the clusters randomly.
3. Assign each point in the training dataset to the closest cluster.
4. Compute the average of the coordinates of every point associated with each class.
5. Re-locate clusters based on the average of the closest data points.
6. Repeat this process until a certain number of iterations is reached, or until the clusters stop moving.
7. Assign a label to the cluster based on majority voting (look at the labels of the trainning point for each cluster).
8. For each test-point, find the cluster cluster and predict label based on the label of the cluster.

2.8 Classifier Fusion

Classifier fusion refers to classifying data using the prediction from multiple classifiers. Two popular classifier fusion techniques are Majority Voting and Naive-Bayes fusion. In this project we implemented the first one. Majority voting classifies data based on the "popular vote" of the algorithms. In other word, outputs from every classifier in the model are taken into consideration, and the class is decided based on what is the most popular prediction between the outputs. Our classifier fusion used three independent classifiers: logistic regression, random forest, and Gaussian-naive bayes.

2.9 Classification Performance Metrics

The performance of the trained model using the corresponding testing data was evaluated. In this section we go over the methods we used to calculate and evaluate the performance of the models. Classification metrics are calculated from true positives (TPs), false positives (FPs), false negatives (FNs) and true negatives (TNs), all of which are tabulated in the so-called confusion matrix in Figure 3. It is important to note that it is not good practice to rely on a single metric. For this reason, more than one metric was used in this project.

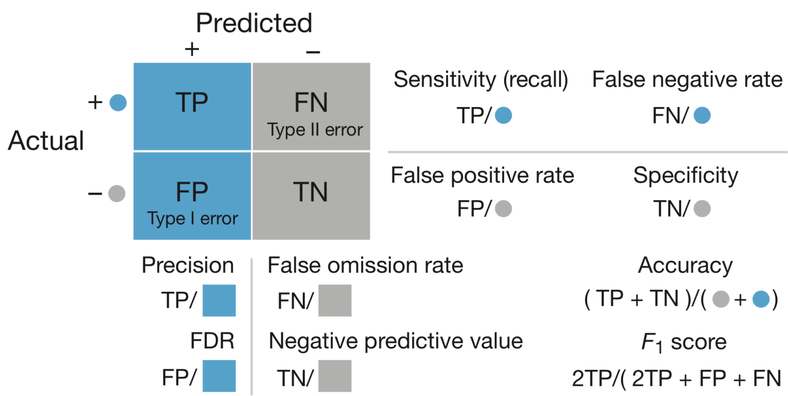


Figure 3: Confusion matrix as efficiently described by Lever et al.: "The confusion matrix shows the counts of true and false predictions obtained with known data. Blue and gray circles indicate cases known to be positive (TP + FN) and negative (FP + TN), respectively, and blue and gray backgrounds/squares depict cases predicted as positive (TP + FP) and negative (FN + TN), respectively. Equations for calculating each metric are encoded graphically in terms of the quantities in the confusion matrix. FDR, false discovery rate" [21].

2.9.1 Numeric Metrics

Accuracy

We report class-wise accuracy by considering True Positive (TP) as the positive reviews which were correctly predicted as positive, and True Negative (TN) as those reviews which were correctly classified as negative. Then, False Positives (FP) and False Negatives (FN) follow intuitively. The actual accuracy of the models were obtained with Equation 9.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}. \tag{9}$$

True Positive Rate/Recall/Hit Rate/Sensitivity

The true positive rate (TPR) measures the proportion of correctly identified actual positives (TP).

$$TPR = \frac{TP}{TP+FN} = 1 - \text{False Negative Rate} \tag{10}$$

False Positive Rate/Fall-Out

The false positive rate (FPR) measures the proportion of incorrectly identified actual negatives (FP).

$$FPR = \frac{FP}{FP + TN} = 1 - \text{Specificity} \tag{11}$$

2.9.2 Graphical Representation of Performance: ROC Curve

ROC is a monotonic [22] probability curve used as a graphical performance measurement for classification problems [23] shown in Figure 4. It uses the TPR and FPR to construct this classifier evaluator.

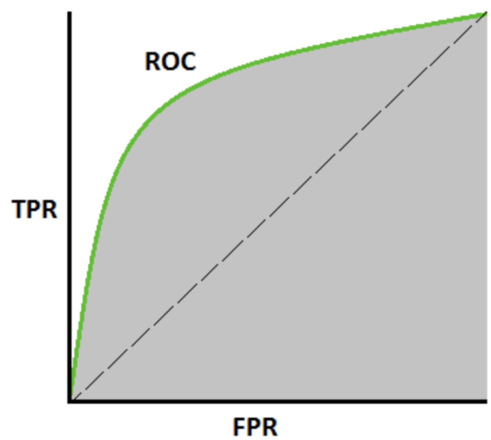


Figure 4: Receiving Operating Characteristic or ROC curve [23]

3 Results

We report the accuracy on each of the 7 classifiers on the four datasets. Confusion matrices are created but only for the Amazon dataset so that the paper is not filled with graphs that look alike. Additionally, we explore the hyper-parameter space for the Gaussian classifier, BPNN, and KNN. For confusion matrices, accuracy tables, and computing time, Gaussian Case I was run with equal prior probabilities, BPNN was designed with an architecture with two hidden layers and 5 neurons in each, and KNN was run with $K = 1$. Finally, K-means did not converge at all during our testing, so we set the number of iterations as 1000. We believe that the reason for this is that our data has too many dimensions, and it is therefore too sparse. We decided not to report confusion matrices for K-means since the algorithm perform poorly. Additionally, we run into problems when training the Gaussian classifier case II and III; when calculating the inverse matrix, the program crashed because of a singular matrix problem. This is a consequence of the dataset, and we did no find a reasonable solution to fix this problem.

3.1 Confusion Matrices

Tables 3 to Table 7 provides information about the class-wise accuracy by separating predictions based on TP, TN, FP, and FN. This tables were created using specifically the Amazon dataset.

3.1.1 Guassian Case I

N = 330	Predicted: Positive	Predicted: Negative
Actual: Positive	125	44
Actual: Negative	66	95

(a) BoW

N = 330	Predicted: Positive	Predicted: Negative
Actual: Positive	131	30
Actual: Negative	49	120

(b) TF-IDF

Table 3: Confusion Matrix for the Gaussian Case I classifier

3.1.2 KNN = 1

N = 330	Predicted: Positive	Predicted: Negative
Actual: Positive	149	20
Actual: Negative	78	83

(a) BoW

N = 330	Predicted: Positive	Predicted: Negative
Actual: Positive	141	28
Actual: Negative	53	108

(b) TF-IDF

Table 4: Confusion Matrix for the KNN = 1 classifier

3.1.3 SVM

N = 330	Predicted: Positive	Predicted: Negative
Actual: Positive	119	50
Actual: Negative	27	134

(a) BoW

N = 330	Predicted: Positive	Predicted: Negative
Actual: Positive	143	18
Actual: Negative	44	125

(b) TF-IDF

Table 5: Confusion Matrix for the SVM classifier

3.1.4 BPNN

N = 330	**Predicted: Positive**	**Predicted: Negative**
Actual: Positive	107	62
Actual: Negative	39	122

(a) BoW

N = 330	**Predicted: Positive**	**Predicted: Negative**
Actual: Positive	126	35
Actual: Negative	26	143

(b) TF-IDF

Table 6: Confusion Matrix for the BPNN classifier

3.1.5 Random Forest

N = 330	**Predicted: Positive**	**Predicted: Negative**
Actual: Positive	121	48
Actual: Negative	31	130

(a) BoW

N = 330	**Predicted: Positive**	**Predicted: Negative**
Actual: Positive	126	35
Actual: Negative	51	118

(b) TF-IDF

Table 7: Confusion Matrix for the Random Forest classifier

3.2 Accuracy

The numeric metric accuracy was chosen to highlight the differences across classifying algorithms.

3.2.1 Classifier Accuracy Comparison: BoW

Table 8 shows the accuracy results of each classifier applied to all four datasets with the BoW features. The number in blue represent the highest accuracy achieved for each dataset. The highest accuracy obtained is 0.803 with SVM on the Amazon dataset, and classifier fusion on the merged dataset. Classifier Fusion was the best performing classifier, which provided an accuracy of 0.778 and 0.787 for the IMDb and Yelp datasets, respectively.

Classifier	Amazon	IMDb	Yelp	Merged
Gaussian (Case I)	0.718	0.613	0.695	0.649
KNN = 1	0.703	0.591	0.636	0.658
SVM	0.803	0.690	0.775	0.794
BPNN	0.66	0.757	0.774	0.677
Random Forest	0.778	0.693	0.756	0.758
K-means	0.487	0.469	0.524	0.520
Classifier Fusion	0.790	0.778	0.787	0.803

Table 8: BoW Accuracy Results.

3.2.2 Classifier Accuracy Comparison: TF-IDF

With TF-IDF features, there seem to be a slight increase in accuracy overall. Again, SVM and Classifier Fusion showed to be the most accurate models. The highest accuracy obtained was 0.837 and it was provided by SVM on the merged dataset. Classifier fusion has performed with a 0.772 accuracy in the IMDb dataset. For the Amazon and Yelp datasets, the highest accuracy obtained were 0.819 and 0.818, respectively.

Classifier	Amazon	IMDb	Yelp	Merged
Gaussian (Case I)	0.789	0.745	0.765	0.771
KNN = 1	0.755	0.712	0.703	0.748
SVM	0.819	0.765	0.818	0.837
BPNN	0.799	0.699	0.748	0.802
Random Forest	0.757	0.646	0.722	0.742
K-means	0.487	0.469	0.524	0.520
Classifier Fusion	0.806	0.772	0.8	0.821

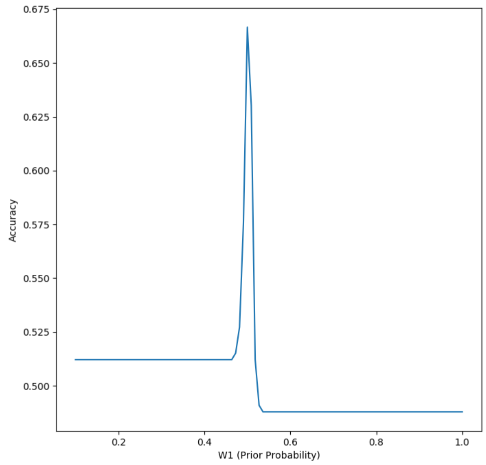
Table 9: TF-IDF Accuracy Results.

3.2.3 Hyper-Parameter Analysis

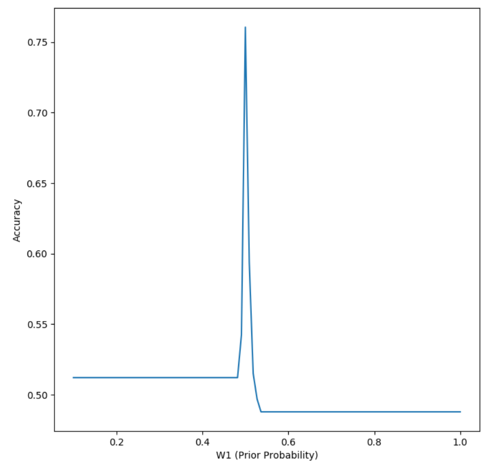
Accuracy was used to evaluate various hyper-parameter studies.

Gaussian Classifier: Prior Probabilities

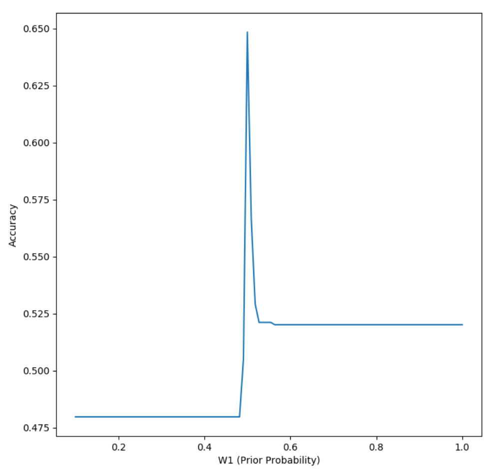
We run the Gaussian classifier (Case I) with different prior probabilities. Given that this paper involves 4 datasets and 2 features extraction methods, we decided to report the results obtained with two datasets and the two distinct features, since the results are consistent across the board. Figure 5 displays the four resulting graphs. Interestingly, all graphs seem to have a peak at one specific prior probability, which provides between 65-79 % accuracy. For all the rest of the priors, the accuracy varies steadily around 50%.



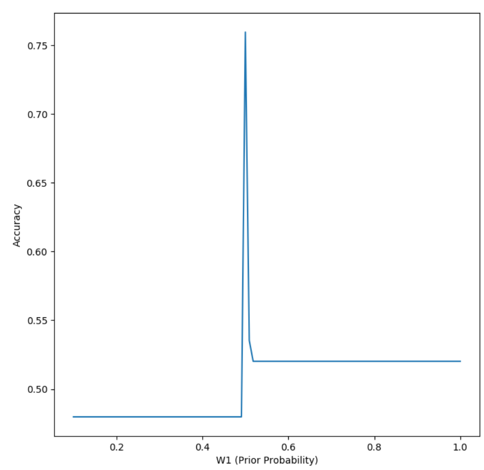
(a) BoW (Amazon dataset)



(b) TF-IDF (Amazon dataset)



(c) BoW (Merged dataset)



(d) TF-IDF (Merged dataset)

Figure 5: Different Prior probabilities for the Gaussian classifier (case I). We used the Amazon dataset and the three-combined dataset.

kNN: K-variation Effect

We run kNN with different Ks to analyse the impact of this hyper-parameter on accuracy. Figure 6 shows the different accuracy scores obtained in with the BoW features. Interestingly, the accuracy seem to peak when $k = 5$ for three of the datasets (Amazon, Yelp, and Merged). For the IMDb dataset, the accuracy seems consistent across the board, with the lowest accuracy value obtained when $k = 5$.

Figure 7 shows the results when different Ks are applied to the TF-IDF dataset. We can clearly see that TF-IDF provides a greater accuracy in general. The accuracy values in this graph seems also very stable, going from around 68% to around 75%.

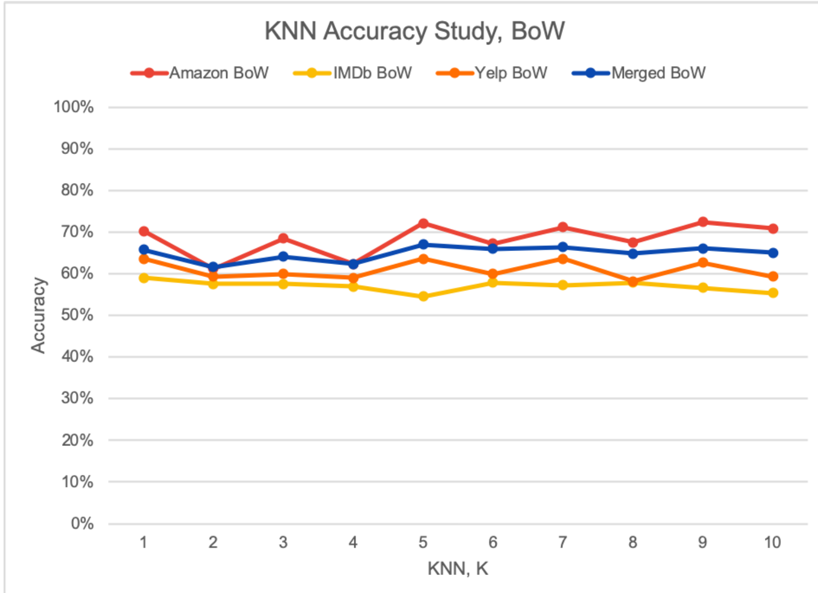


Figure 6: KNN accuracy study for different K values using BoW

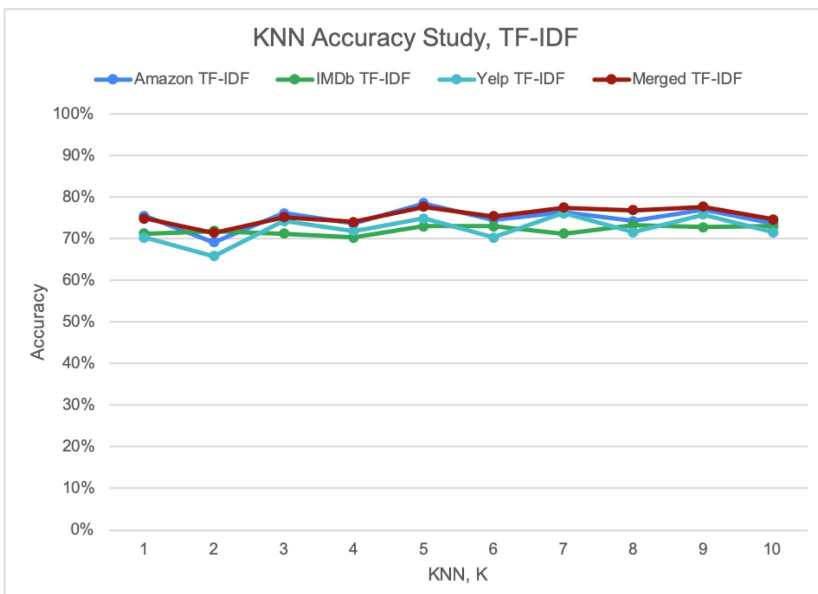
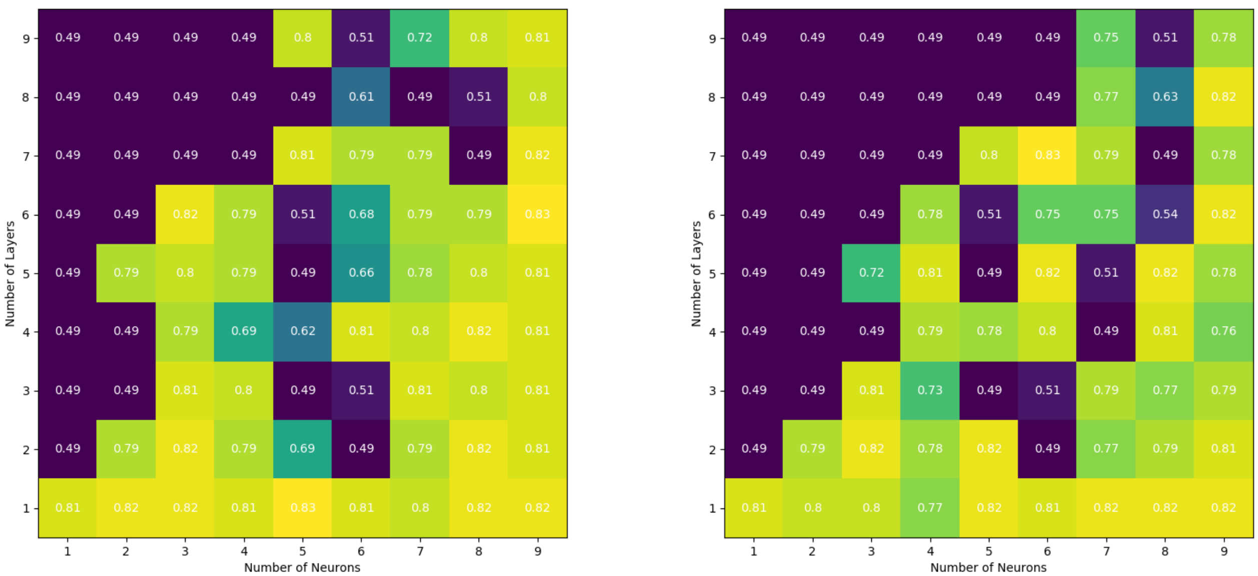


Figure 7: KNN accuracy study for different K values using TF-IDF

BPNN: Neurons vs Layers Analysis

When implementing a Neural Network, it is often challenging to select the appropriate architecture (number of layers, and number of neurons in each layer). We coded a function that test 100 defines and trains 100 different architectures, and then computes the accuracy for the Amazon dataset (BoW and TF-IDF). The loop was creates so that the number of hidden layers goes from 1 to 10, and the number of neurons in each layer also goes from 1 to 10. Note that the number of neurons will be the same in each hidden layer, in other words, if the number of layers is 3 and the number of neurons is 4, the architecture will look like: [4,4,4]. Figure 8 shows that in general, a large number of neurons increases the accuracy of the model. A large number of hidden layers will increase the accuracy only if the each layer contains a relatively large number of neurons.



(a) BoW (b) TF-IDF

Figure 8: Number of Layers vs Number of Neurons. This graph was created using the Amazon dataset only. The numbers within the square show the accuracy when BPNN is trained with the specified architecture

3.3 Computational Cost

The computational cost to train and evaluate each of the classifier is shown in Figure 9 to Fig. 14. Even though SVM has reported really high accuracy rates, it has also been the most computational expensive algorithm. Surprisingly, Classifier fusion was significantly cheaper in terms of computational cost.

BoW	Computational Cost in Seconds			
	Amazon	IMDb	Yelp	merged
Gaussian Case I	1.93	4.91	2.61	44.80
KNN = 1	1.04	1.37	1.07	18.85
SVM	19.72	36.96	21.12	504.07
BPNN (5,5)	6.32	14.86	17.06	83.31
Randon Forest	1.08	1.72	1.14	14.96
Classifier fusion	0.55	0.77	0.44	3.94

Figure 9: Computational Cost Data for BoW

TF-IDF	Computational Cost in Seconds			
	Amazon	IMDb	yelp	merged
Gaussian Case I	1.80	4.32	2.42	28.54
KNN = 1	0.80	1.04	0.84	16.30
SVM	21.37	36.34	22.60	710.45
BPNN (5,5)	3.25	8.92	7.71	34.21
Randon Forest	1.04	1.76	1.26	11.09
Classifier fusion	0.46	0.65	0.40	3.74

Figure 10: Computational Costs Data for TF-IDF

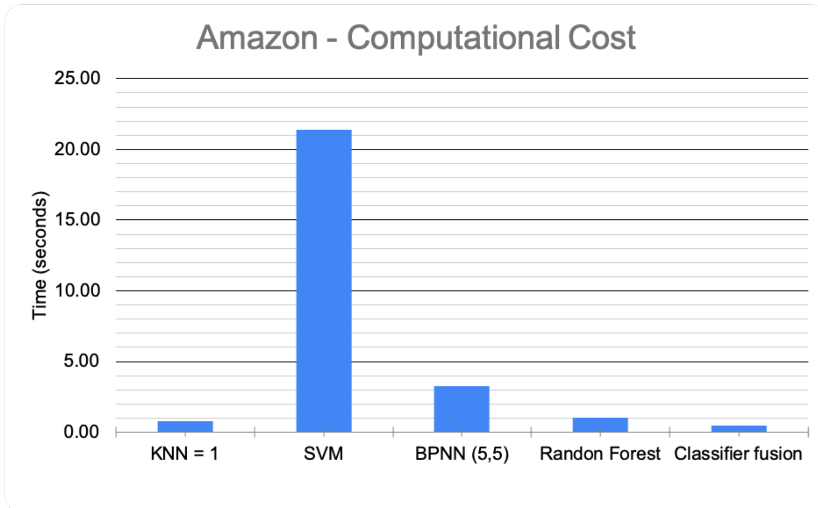


Figure 11: Computational Cost: Amazon Dataset

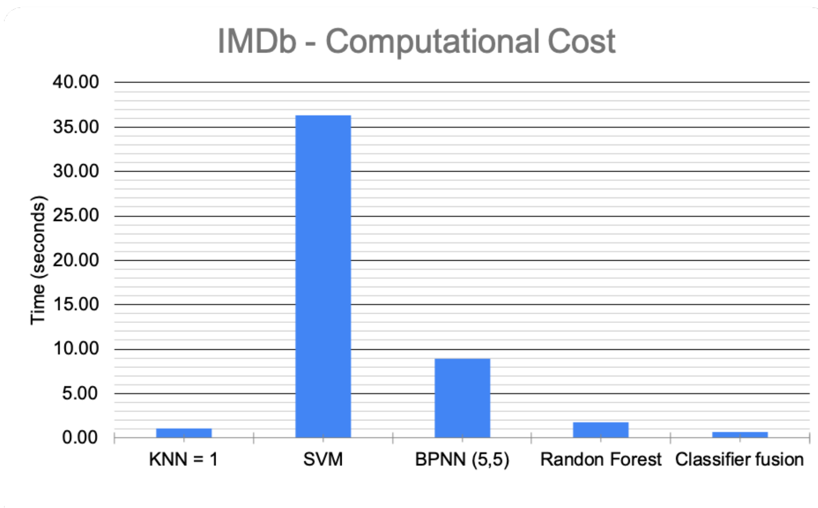


Figure 12: Computational Cost: IMDb Dataset

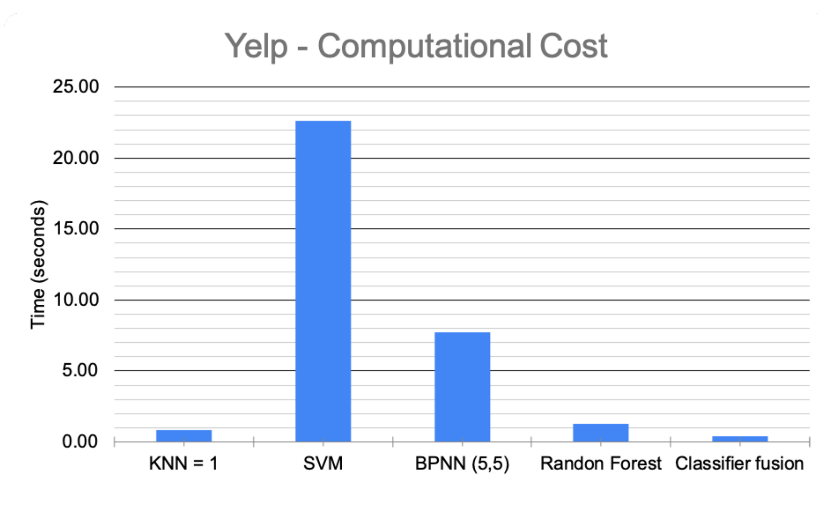


Figure 13: Computational Cost: Yelp

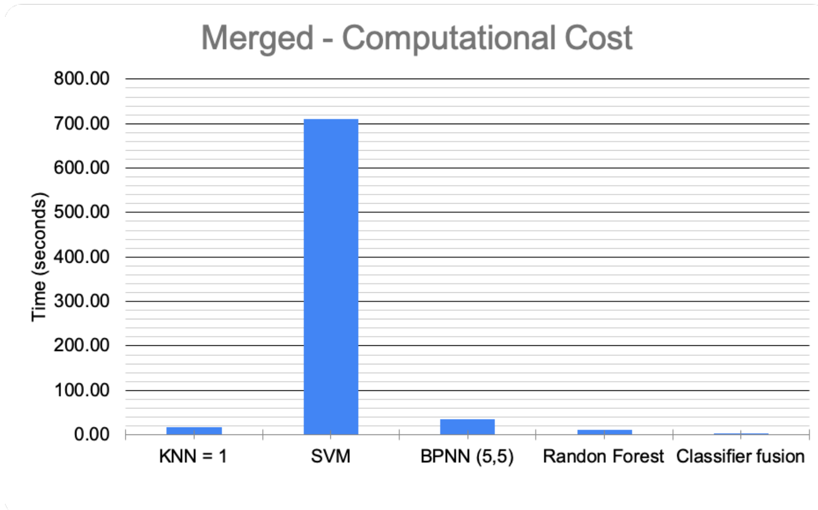


Figure 14: Computational Cost: Merged Dataset

4 Discussion

Natural Language Processing has received special attention during the last couple of years. More precisely, the task of sentimental analysis is a hot area specially because of all the possible commercial applications. As it often occurs in ML and AI, most of the work developed in NLP is still extremely specialized, since algorithms have shown to perform differently in a variety of text data. Nevertheless, as the frontiers of NLP are expanded, humanity gets one step closer to the fascinating idea of teaching computers how to independently communicate. Today, most of the cutting edge work is done with complex feature extraction techniques such as Word2vec that can be fed into deep learning models. However, classic feature extraction techniques such as BoW and TF-IDF have proved to be success with high accuracy levels and they are still used in nuemrous applications. This project focused on implementing traditional ML models and comparing them with the results provided by the authors of the dataset [17]. Kotznias et al. developed a DL model and reported acuracies between 86-89 %. In this project, we showed that using SVM and Classifier Fusion, we were able to obtain up to 83% of accuracy. The authors do not provide information about computational cost, so accuracy is the only metric that was compared. In addition to overall accuracy, this paper presented confusion matrices, computational cost tables, and an analysis of hyper-parameters. Interestingly, Classifier Fusion has provided high accuracy scores with relatively low computing cost. Support Vector Machine also showed high accuracy in contrast with the other algorithms, but it has also been expensive in terms of computational cost. Future work could include more algorithms and newer feature extraction techniques. Additionally, it would be interesting to analyze the effect of training an algorithm with the Amazon dataset, and test it with the Yelp dataset, for example. If succeeded, we could easily train algorithms with dataset already existent, and apply them to situation or tasks where datasets are still scarce.

References

[1] M. Hu and B. Liu, "Mining and summarizing customer reviews,"

[2] P. D. Turney, "Thumbs up or thumbs down? semantic orientation applied to unsupervised classification of reviews,"

[3] J. W. Theresa Wilson and P. Hoffmann, "Recognizing contextual polarity in phrase-level sentiment analysis,"

[4] Elvis, "Fix this please https://medium.com/dair-ai/detecting-sarcasm-with-deep-convolutional-neural-networks-4a0657f79e80,"

[5] H. W. Abram Handler, Matthew J. Denny and B. O'Connor, "Bag of what? simple noun phrase extraction for text analysis,"

[6] J. Ramos, "Using tf-idf to determine word relevance in document queries,"

[7] H. J. S. S. D. Y. S. B. K. P. E. P. R. L. V. R. K. D. Dekang Lin, Kenneth Church and S. Narsale, "New tools for web-scale n-grams,"

[8] B. C. P Majumder, M Mitra, "N-gram: a language independent approach to ir and nlp,"

[9] J. H. M. Daniel Jurafsky, "Speech and language processing,"

[10] O. Levy and Y. Goldberg, "Dependency-basedword embeddings,"

[11] N. N. I. Santos and L. de Macedo Mourelle, "Sentiment analysis using convolutional neural network with fasttext embeddings," *2017 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*, pp. 1–5, 2017.

[12] C. N. dos Santos and M. Gatti, "Deep convolutional neural networks for sentiment analysis of short texts,"

[13] T. K. Martin Wöllmer, Felix Weninger and T. Björn Schuller, "Youtube movie reviews: Sentiment analysis in an audiovisual context,"

[14] J.-C. N. Tun Thura Thet and C. S. Khoo, "Aspect-based sentiment analysis of movie reviews on discussion boards,"

[15] G. F. Dietmar Gräbnera, Markus Zankerb and M. Fuchs, "Classification of customer reviews based on sentiment analysis,"

[16] G. T. Q. P. Raiyani, Kashyap and V. Nogueira, "Fully connected neural network with advance preprocessor to identify aggression over facebook and twitter,"

[17] N. D. F. Dimitrios Kotzias, Misha Denil and P. Smyth1, "From group to individual labels using deep features,"

[18] D. G. S. Richard O. Duda, Peter E. Hart, *Pattern Classification*. 2nd ed.

[19] M. Nielsen, "Neural networks and deep learning,"

[20] R. S. Brid, "Introduction to decision trees." https://medium.com/greyatom/decision-trees-a-simple-way-to-visualize-a-decision-dc506a403aeb, 2008. [Online; accessed 5-December-2009].

[21] J. Lever, M. Krzywinski, and N. Altman, "Points of significance: Logistic regression," *Nature Methods*, vol. 13, pp. 541–542, 6 2016.

[22] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2 ed., November 2000.

[23] S. Narkhede, "Understanding auc - roc curve," May 2019.

5 Appendix

5.1 Python Script

5.1.1 main.py

```python
'''
Created by Kevin De Angeli & Hector D. Ortiz-Melendez
Date: 2019-11-24
'''

from GaussianClassifiers import *
from KNN import *
import matplotlib.pyplot as plt
from K_means import *

import time #For computing time
from sklearn.metrics import confusion_matrix #To compute the confusion matrix.
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer #Bag-of-words/TF-IDF (Feature Extraction)
from sklearn import svm #Support Vector Machine
from sklearn.model_selection import train_test_split #To split the data into testing/training
from sklearn.metrics import accuracy_score #To compute the accuracy of each model
from sklearn import tree #Decision Trees
from sklearn.neural_network import MLPClassifier #For BPNN
from sklearn.model_selection import KFold #to split the data
from sklearn.ensemble import RandomForestClassifier #RnadomForest
from sklearn.linear_model import LogisticRegression #For Classifier fussion
from sklearn.naive_bayes import GaussianNB # For classifier fussion
from sklearn.ensemble import VotingClassifier  #For classifier fussion.
from sklearn.utils import shuffle #To shuffle the data when we merge the three subsets.

def readData(path):
    file = open(path, "r")
    Y = []
    X = []
    while True:
        line = file.readline()
        if not line:
            break
        X.append(line[0:-3])
        Y.append(line[-2])
    Y = [int(i) for i in Y]  # make labels int instead of str

    return np.array(X), np.array(Y)
```

```

def getDataInfo(X, labels = False):
    print("***********************")
    print("Example first 3 rows: \n")
    print(X[0:3])
    print("\nNumber of elements: ", len(X))
    if labels:
        print("Number of entries with label 1: ", np.sum(X==1))
        print("Number of entries with label 0: ", len(X)-np.sum(X==1))
    print("***********************\n")



def BoW(X):

    vectorizer = CountVectorizer()
    Xv = vectorizer.fit_transform(X)
    X_bow = Xv.toarray()
    return X_bow

def TF_IDF(X):
    vectorizer = TfidfVectorizer()
    Xv = vectorizer.fit_transform(X)
    X_TFIDF = Xv.toarray()
    return X_TFIDF

def splitData(X,y,testSize=0.33):
    #This is the most basic way to split data:  77% Trainning and 33% testing
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=testSize,
    ↪ random_state=42)
    return X_train, X_test, y_train, y_test

def SVM(X_train, X_test, y_train, y_test):
    clf = svm.SVC(gamma='scale')
    clf.fit(X_train, y_train)
    y_guess = clf.predict(X_test)
    # get support vectors
    clf.support_vectors_
    # get indices of support vectors
    clf.support_
    # get number of support vectors for each class
    clf.n_support_
    return accuracy_score(y_test, y_guess)

def BPNN(X_train, X_test, y_train, y_test, hiddenLayer = (5,2)):
```

```python
    clf = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=hiddenLayer, random_state=1)
    clf.fit(X_train, y_train)
    y_guess = clf.predict(X_test)
    [coef.shape for coef in clf.coefs_]
    return accuracy_score(y_test, y_guess)

def DecisionTree(X_train, X_test, y_train, y_test):
    clf = tree.DecisionTreeClassifier()
    clf.fit(X_train, y_train)
    y_guess = clf.predict(X_test)
    return accuracy_score(y_test, y_guess)

def gaussian(data):
    X, Y = readData(data)
    Xv1 = BoW(X)
    X_train, X_test, y_train, y_test = splitData(Xv1, Y)
    model = mpp(2)
    model.fit(X_train, y_train)
    prediction = model.predict(X_test)
    print("Gaussian Accuracy:",accuracy_score(y_test, prediction))

def threeVsAll(data):
    #This is the function of Milestone 2 used to compute some initial results.
    for dat in data:
        X, Y = readData(dat)
        Xv1 = BoW(X)
        Xv2 = TF_IDF(X)
        featEx = [Xv1, Xv2, 'BoW', 'TF_IDF']
        count = 1
        for Xv in featEx[0:2]:
            count += 1
            X_train, X_test, y_train, y_test = splitData(Xv, Y)
            accSVM = SVM(X_train, X_test, y_train, y_test)
            accBPNN = BPNN(X_train, X_test, y_train, y_test)
            accDT = DecisionTree(X_train, X_test, y_train, y_test)
            print('Data:', dat, '\nFeature Extraction:', featEx[count], '\nSVM', accSVM, 'BPNN', accBPNN, 'DT', accDT,
                  "\n")
    #    pdb.set_trace()


def crossValidationExample(X,Y, classifierClass):
    #This is an example of how you do k-fold cross validation.
    #X, Y = readData(data)
    timeStart = time.time()
```

```
    X = BoW(X)
    kf = KFold(n_splits=10) # Split the data into 10 subsets.
    kf.get_n_splits(X)
    accuracy = []
    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = Y[train_index], Y[test_index]
        clf = classifierClass
        #clf = tree.DecisionTreeClassifier()
        clf.fit(X_train, y_train)
        y_predict = clf.predict(X_test)
        accuracy.append(accuracy_score(y_test, y_predict))
    print("--- %s seconds ---" % (time.time() - timeStart))
    #print(accuracy)
    print(np.mean(accuracy))



def randomForest(X_train, X_test, y_train, y_test):
    clf = RandomForestClassifier( random_state=0)
    clf.fit(X_train, y_train)
    y_predict = clf.predict(X_test)
    print(accuracy_score(y_test, y_predict))

def mergeDatasets(data):
    x1, y1 = readData(data[0])
    x2, y2 = readData(data[1])
    x3, y3 = readData(data[2])
    X_all = np.concatenate([x1,x2,x3])
    Y_all = np.concatenate([y1,y2,y3])
    X_all, Y_all = shuffle(X_all, Y_all, random_state=0) #Shuffle data
    return X_all, Y_all

def plotConfusionMatrix(y_predict, y_test):
    plt.figure(num=None, figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
    labels = [1, 0]
    cm = confusion_matrix(y_test, y_predict, labels)
    print(cm)
    print(type(cm))
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(cm)
    #plt.title('Confusion matrix of the classifier')
    fig.colorbar(cax)
    ax.set_xticklabels([''] + labels)
    ax.set_yticklabels([''] + labels)
```

```
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.show()

def plotAmazonCM():
    plt.figure(num=None, figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
    labels = [1, 0]
#     cm = confusion_matrix(y_test, y_predict, labels)

    # Amazon
    AmazonCMbow = [83, 78, 20, 149]

    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(AmazonCMbow)
    #plt.title('Confusion matrix of the classifier')
    fig.colorbar(cax)
    ax.set_xticklabels([''] + labels)
    ax.set_yticklabels([''] + labels)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.show()

    AmazonCMtf = [108, 53, 30, 139]
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(AmazonCMtf)
    #plt.title('Confusion matrix of the classifier')
    fig.colorbar(cax)
    ax.set_xticklabels([''] + labels)
    ax.set_yticklabels([''] + labels)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.show()


def NeuronsVSLayersVsAccuracy3D(X_train, X_test, y_train, y_test):
    #Note: The array of neurons and the array of hl should be the same
    #The program can be modified so it can take arbitrary numbers.
    neurons = np.arange(1, 10)  # Controls number of neurons in all layers.
    hl = np.arange(1,10)  # Controls number of layers
    network = []
    #network.append(784)
    #test = list(test_data)
    #network.append(1)
    accuracy =  np.zeros([hl.shape[0],neurons.shape[0]])
```

```
    for i in hl:
        network.append(1)
        for n in neurons:
            for k in range(len(network)):
                network[k] = n
            print("Network Architecture Being used: ",network)
            clf = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=tuple(network), random_state=1)
            clf.fit(X_train, y_train)
            y_guess = clf.predict(X_test)
            [coef.shape for coef in clf.coefs_]
            acc= accuracy_score(y_test, y_guess)
            acc = np.round_(acc,decimals=2)
            accuracy[hl.shape[0]-i][n-1] = acc


    print(accuracy)

    #Note: The heat map fuction displays the graph in the exact order as the
    #Matrix. So I had to populate the matrix in the opposite order
    left = neurons[0] - .5  # Should be set so that it starts a the first point in the array -.5
    right = neurons[-1] + .5  # last number of the array +.5
    bottom = hl[0] - .5
    top = hl[-1] + .5
    extent = [left, right, bottom, top]

    fig, ax = plt.subplots(figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
    im = ax.imshow(accuracy, extent=extent, interpolation='nearest')
    ax.set(xlabel='Number of Neurons', ylabel='Number of Layers')

    #Label each square in the heat map:
    for i in range(len(hl)):
        for j in range(len(neurons)):
            text = ax.text(j + 1, i + 1, accuracy[accuracy.shape[0]-i-1,j],
                           ha="center", va="center", color="w")

    plt.show()


def classifierFussion(data):
    clf1 = LogisticRegression(random_state=1)
    clf2 = RandomForestClassifier(n_estimators=50, random_state=1)
    clf3 = GaussianNB()
    for dat in data:
        X, Y = readData(dat)
```

```
        Xv1 = BoW(X)
        Xv2 = TF_IDF(X)
        featEx = [Xv1, Xv2, 'BoW', 'TF_IDF']
        count = 1
        for Xv in featEx[0:2]:
            count += 1
            X_train, X_test, y_train, y_test = splitData(Xv, Y)
            eclf1 = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)], voting='hard')
            timeStart = time.time()
            eclf1 = eclf1.fit(X_train, y_train)
            y_predict = eclf1.predict(X_test)

            print('Data:', dat, '\nFeature Extraction:', featEx[count],accuracy_score(y_test, y_predict))
            print("--- %s seconds ---" % (time.time() - timeStart))
            print(" ")

    X, Y = mergeDatasets(data)
    Xv1 = BoW(X)
    X_train, X_test, y_train, y_test = splitData(Xv1, Y)
    eclf1 = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)], voting='hard')
    timeStart = time.time()
    eclf1 = eclf1.fit(X_train, y_train)
    y_predict = eclf1.predict(X_test)
    print('Data: All-Three', '\nFeature Extraction:BoW', accuracy_score(y_test, y_predict))
    print("--- %s seconds ---" % (time.time() - timeStart))
    print(" ")

    Xv2 = TF_IDF(X)
    X_train, X_test, y_train, y_test = splitData(Xv2, Y)
    eclf1 = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)], voting='hard')
    timeStart = time.time()
    eclf1 = eclf1.fit(X_train, y_train)
    y_predict = eclf1.predict(X_test)
    print('Data: All-Three', '\nFeature Extraction:= TF-IDF', accuracy_score(y_test, y_predict))
    print("--- %s seconds ---" % (time.time() - timeStart))
    print(" ")



def gaussianPriorProbsAnalysis(data, bow = True):
```

```
    if len(data) == 3:
        X, Y = mergeDatasets(data)  # all three combined
    else:
        X, Y = readData(data)  # one dataset at the time

    if bow:
        X = BoW(X)
    else:
        X= TF_IDF(X)

    X_train, X_test, y_train, y_test = splitData(X, Y)
    priors=np.linspace(0.1, 1, 100) #log(0) DNE
    ws = [0,0]
    acc = []
    for w in priors:
        ws[0] = w
        ws[1] = 1-w+.000001
        gaussian = mpp(1)
        gaussian.pw = ws
        gaussian.fit(X_train, y_train)
        prediction = gaussian.predict(X_test)
        acc.append(accuracy_score(y_test, prediction))

    print(acc)
    plt.figure(num=None, figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
    plt.plot(priors, acc)
    plt.xlabel(xlabel='W1 (Prior Probability)')
    plt.ylabel(ylabel='Accuracy')
    plt.savefig('Images/priorProbs/example.png')

    #plt.show()

def knnStudy(dat, X, Y, ks, ke, merged=False):
        Xv1 = BoW(X)
        Xv2 = TF_IDF(X)
        featEx = [Xv1, Xv2, 'BoW', 'TF_IDF']
        count = 1

        print('TPR', 'FPR')

        for Xv in featEx[0:2]:
            count += 1
            X_train, X_test, y_train, y_test = splitData(Xv, Y)
#             accSVM = SVM(X_train, X_test, y_train, y_test)
#             accBPNN = BPNN(X_train, X_test, y_train, y_test)
#             accDT = DecisionTree(X_train, X_test, y_train, y_test)
```

```
            if merged == False:
                print('Data:', dat, '\nFeature Extraction:', featEx[count],"\n")
            else:
                print('Data: Merged Data ', '\nFeature Extraction:', featEx[count],
                 ↪  '\nKNN', "\n")


            for k in range(ks,ke+1):

                start = time.time()

                Acc, TPR, TNR = knn(X_train, X_test, y_train, y_test, k)

                end = time.time()


def showConfusionMatrix(data, classifierClass):
    X, Y = readData(data)  # one dataset at the time
    labels = [1, 0]

    #X = TF_IDF(X)

    print("BOW")
    X = BoW(X)
    X_train, X_test, y_train, y_test = splitData(X, Y)

    classifier = classifierClass
    classifier.fit(X_train,y_train)
    y_predict=classifier.predict(X_test)
    cm = confusion_matrix(y_test, y_predict, labels)
    #Output is in this format: tn, fp, fn, tp
    print(cm)
    print(" ")
    print("TFIDF")
    X, Y = readData(data)  # one dataset at the time
    X = TF_IDF(X)
    X_train, X_test, y_train, y_test = splitData(X, Y)
    classifier = classifierClass
    classifier.fit(X_train, y_train)
    y_predict = classifier.predict(X_test)
    cm = confusion_matrix(y_test, y_predict)
    # Output is in this format: tn, fp, fn, tp
    print(cm)
```

```

##########################################################################
###################### Main Starts Here  #################################
##########################################################################

def main():
    amazon= 'Data/amazon_cells_labelled.txt'
    imdb= 'Data/imdb_labelled.txt'
    yelp = 'Data/yelp_labelled.txt'
    data = [amazon, imdb, yelp]


    #threeVsAll(data) # Function from Milestone 3 to compute initial results
    #gaussian(imdb) #Case 1 works and gives bad accuracy (50%). Case II, and III don't work because of singular matrix when taking the inverse.
    #crossValidationExample(amazon, classifierClass=tree.DecisionTreeClassifier()) #Give the dataset and the classifier ;)
    #gaussianPriorProbsAnalysis(data, False) #False for TFIDF, True for BoW
    #classifierFussion(data)


    '''
    #This is to make the accuracy tables. Make sure to check what type of
    #features are being used in the function crossValidationExample (bow or tf-dif).

    #X,Y = readData(yelp) # one dataset at the time
    X, Y = mergeDatasets(data) #all three combined
    crossValidationExample(X=X,Y=Y, classifierClass=RandomForestClassifier(random_state=0)) #Give the dataset and the classifier ;)




    #This was used to plot the confusion matrix

    X,Y = readData(amazon) # one dataset at the time
    #X, Y = mergeDatasets(data) #all three combined
    X = BoW(X)
    #X = TF_IDF(X)
    X_train, X_test, y_train, y_test = splitData(X, Y)
    clf = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(5,5), random_state=1)
    clf.fit(X_train, y_train)
    prediction = clf.predict(X_test)
    plotConfusionMatrix(prediction, y_test)
```

```python
    '''
    #This code is to create the heat map with the neuronsVsLayers.

#     X,Y = readData(amazon) # one dataset at the time
#     X = TF_IDF(X)
#     X_train, X_test, y_train, y_test = splitData(X, Y)
#     NeuronsVSLayersVsAccuracy3D(X_train, X_test, y_train, y_test)

    kstart  = 1
    kend    = 10
    # KNN individual data
    for dat in data:
        X, Y = readData(dat)
        knnStudy(dat, X, Y, kstart, kend)

    # KNN merged data
    X, Y = mergeDatasets(data)
    knnStudy('ignore', X, Y, kstart, kend, True)

    '''
    #This code runs KNN

    X,Y = readData(yelp) # one dataset at the time
    #X, Y = mergeDatasets(data) #all three combined
    X = TF_IDF(X)
    X_train, X_test, y_train, y_test = splitData(X, Y)
    k_means = Kmeans()
    k_means.fit(X_train , y_train, iterationsLimit= 1000)
    y_predict = k_means.predict(y_test)
    print(accuracy_score(y_test, y_predict))
    '''

    #showConfusionMatrix(amazon, svm.SVC(gamma='scale'))
    #showConfusionMatrix(amazon, MLPClassifier(solver='lbfgs', alpha=1e-5,
    ↪  hidden_layer_sizes=(5,5), random_state=1))
    #showConfusionMatrix(amazon, RandomForestClassifier( random_state=0))
    '''
    clf1 = LogisticRegression(random_state=1)
    clf2 = RandomForestClassifier(n_estimators=50, random_state=1)
    clf3 = GaussianNB()
    showConfusionMatrix(amazon,  VotingClassifier(estimators=[('lr', clf1), ('rf',
↪   clf2), ('gnb', clf3)], voting='hard'))
```

```
    '''
    showConfusionMatrix(amazon,  mpp())




if __name__ == "__main__":
    main()
```

5.1.2 Kmeans.py

```
'''
Created by Kevin De Angeli
Date: 2019-12-07
'''

import copy
import numpy as np
import time #For computing time


class Kmeans:
    def __init__(self, k=2):
        self.k=k
        self.C = []
        self.iterations = 0
        self.X = []
        self.ClusterClass = np.array([-1,-1])
        self.Y =[]


    def fit(self, x_train, y_train, iterationsLimit=-1):
        timeStart = time.time()

        self.k = np.unique(y_train).shape[0]
        vectorDimensions = x_train[0].shape[0]

        self.C.append(x_train[7])
        self.C.append(x_train[7])
        #self.C = [np.random.randint(low=0, high=1, size=vectorDimensions) for i in range(self.k)]
        self.X = x_train
        self.C = np.array(self.C)
        self.Y = y_train


        C_old = np.array([])
```

```python
        while not self.finishLoopCheck(oldClusters=C_old, iterationsLim=iterationsLimit):
            print("Iteration: ", self.iterations)
            C_old = copy.deepcopy(self.C)  # To copy C by value not by reference
            dataAssignment = self.closestCluster()
            self.clustersUpdate(dataAssignment)
            self.iterations += 1

        if iterationsLimit == self.iterations:
            clusterAssignment = []
            for i in self.X:  # For each dataPoint
                dist = []
                for k in self.C:  # For each cluster.
                    dist.append(np.linalg.norm(i - k))
                min = np.amin(dist)
                index = dist.index(min)
                clusterAssignment.append(index)

            clusterAssignment=np.array(clusterAssignment)
            ClusterOne = clusterAssignment == 1
            countPositives = np.sum(self.Y[ClusterOne])
            print(countPositives)
            if countPositives >= (clusterAssignment.shape[0]/2):
                self.ClusterClass[0]=1
                self.ClusterClass[1]=0
            else:
                self.ClusterClass[0]=0
                self.ClusterClass[1]=1
        print("--- %s seconds ---" % (time.time() - timeStart))

    def predict(self, y_test):
        self.X= y_test

        clusterAssigned = self.closestCluster()
        classifyAccordingly = clusterAssigned == 0
        predictions = copy.deepcopy(clusterAssigned)  # To copy C by value not by reference
        predictions[classifyAccordingly] = self.ClusterClass[0]
        classifyAccordingly = clusterAssigned == 1
        predictions[classifyAccordingly] = self.ClusterClass[1]
        return predictions
```

```
    def reInitializeEmptyClusters(self, CIndex):
        '''
        Re-initialize clusters at randon.
        This is used when clusters are empty.
        '''

        newCoordinates = np.random.randint(low=0, high=256, size=self.X.shape[1])
        self.C[CIndex] = np.array(newCoordinates)

    def clustersUpdate(self, clusterAssignments):
        '''
        In order to handle "empty clusters" I re-initialized those clusters randonly.
        '''
        # clusterAssignments = np.array(clusterAssignments)
        newClusterCoordinate = []
        # update self.C based on clusterAssignments

        for i in range(self.C.shape[0]):
            if i not in clusterAssignments:
                print("Empty Cluster: ", i)
                self.reInitializeEmptyClusters(CIndex=i)
                continue
            findDataPoints = clusterAssignments == i

            dataPointsCoordinates = self.X[findDataPoints]
            newClusterCoordinate = np.average(dataPointsCoordinates, axis=0)
            self.C[i] = newClusterCoordinate



    def finishLoopCheck(self, oldClusters, iterationsLim):
        '''
        Stop the program if the clusters' position stop changing or
        the limit number of iterations has been reached.
        '''
        if iterationsLim == self.iterations:
            return True
        else:
            return np.array_equal(oldClusters, self.C)  # Clusters didn't change ?

    def closestCluster(self):
        '''
        Create a list where each data point is associated with a
        clusters. Then it returns the list of clusters.

```

```
        '''
        clusterAssignment = []
        for i in self.X:  # For each dataPoint
            dist = []
            for k in self.C:  # For each cluster.
                dist.append(np.linalg.norm(i - k))
            min = np.amin(dist)
            index = dist.index(min)
            clusterAssignment.append(index)

        # return a list of size X where each element specifies the cluster.
        return np.array(clusterAssignment)
```

5.1.3 KNN.py

```
'''
Created by Kevin De Angeli
Date: 2019-12-03
'''

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from statistics import mean
#from scipy import integrate
from numpy import array
from numpy import cov
#from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt
import sympy as sym
from sympy.matrices import MatrixSymbol, Transpose
from sympy.functions import transpose
from sympy import symbols, Eq, solve, nsolve
import math
from mpmath import *
import pdb
import matplotlib.cm as cm
import matplotlib.cbook as cbook
from matplotlib.path import Path
from matplotlib.patches import PathPatch
from Distances import *

import operator
import time

def knn(nXtrain, nXtest, y_train, y_test, k):
```

```
    A0 = 1
    A1 = 1

    knn         = []

    TP = 0
    TN = 0
    FP = 0
    FN = 0

    rowTe       = len(nXtest)
    columns     = len(nXtest[0])-1

    #    classes     = nXtest[:,-1]
    #    guesses     = nXtrain[:,-1]
    classes     = y_test
    guesses     = y_train


    for row in range(rowTe):


        distances       = []

        for x in range(len(nXtrain)):

            diff = nXtest[row] - nXtrain[x]

            dist = np.dot(diff, diff)

            distances.append((dist,guesses[x]))


        #print(distances)
        distances.sort(key=operator.itemgetter(0))

        for K in range(k):
            knn.append(distances[K][1])


        label0 = 0
        label1 = 0

        for K in knn:
```

```
            if K == 0:
                label0 += 1
            else:
                label1 += 1

        if (A0 == 1 and A1 == 1):
            if label0 > label1:
                guess = 0
            else:
                guess = 1
        else:
            # prior chosen
            ca0      = label0 * A0 / k
            ca1      = label1 * A1 / k

            if ca0 > ca1:
                guess = 0
            else:
                guess = 1


        if   (guess == 1 and classes[row] == 1):
            # True Positive
            TP += 1

        elif (guess == 0 and classes[row] == 1):
            # False Negative
            FN += 1

        elif (guess == 1 and classes[row] == 0):
            # False Positive
            FP += 1

        elif (guess == 0 and classes[row] == 0):
            # True Negative
            TN += 1


        knn     = []


    Acc = (TP + TN) / (TP + TN + FP + FN)
    TPR = TP / (TP + FN)     # sensitivty
    FPR = FP / (FP + TN)

    TNR = 1 - FPR            # specificity
```

```
    FNR = 1 - TPR

    #    print('CM: TN, FP, FN, TP = ', TN, FP, FN, TP)

    #    print('kNN Accuracy: ' , Acc)
    #    print('TPR,TNR',TPR,TNR)

    '''
#    print('CM: TN, FP, FN, TP = ', TN, FP, FN, TP)
#    print('K:',k,'Accuracy:' , Acc)
#    print('sensitivity,specificity',TPR,TNR)
    '''
    print(TPR,FPR)


#    if (A0 == 1 and A1 == 1):
#       if k == 1:
#           print('CM: TN, FP, FN, TP = ', TN, FP, FN, TP)
#           print('K:',k,'Accuracy:' , Acc)
#           print('sensitivity,specificity',TPR,TNR)
#       elif k == 10:
#           print('CM: TN, FP, FN, TP = ', TN, FP, FN, TP)
#           print('K:',k,'Max Accuracy:' , Acc)
#           print('sensitivity,specificity',TPR,TNR)



    return Acc, TPR, TNR
```

5.1.4 GaussianClassifiers.py

```
import numpy as np
from Distances import *


class mpp:
   def __init__(self, case=1):
      # init prior probability, equal distribution
      # self.classn = len(self.classes)
      # self.pw = np.full(self.classn, 1/self.classn)

      # self.covs, self.means, self.covavg, self.varavg = \
      #     self.train(self.train_data, self.classes)
      self.case_ = case
      self.pw_ = [0.5,0.5]

   def fit(self, Tr, y):
```

```
        # derive the model
        self.covs_, self.means_ = {}, {}
        self.covsum_ = None

        self.classes_ = np.unique(y)  # get unique labels as dictionary items
        self.classn_ = len(self.classes_)

        for c in self.classes_:
            arr = Tr[y == c]
            self.covs_[c] = np.cov(np.transpose(arr))
            self.means_[c] = np.mean(arr, axis=0)  # mean along rows
            if self.covsum_ is None:
                self.covsum_ = self.covs_[c]
            else:
                self.covsum_ += self.covs_[c]

        # used by case II
        self.covavg_ = self.covsum_ / self.classn_

        # used by case I
        self.varavg_ = np.sum(np.diagonal(self.covavg_)) / len(self.classes_)

    def predict(self, T):
        # eval all data
        y = []
        disc = np.zeros(self.classn_)
        nr, _ = T.shape


        if self.pw_ is None:
            self.pw_ = np.full(self.classn_, 1 / self.classn_) #Equal Prior probability
        #print(self.pw)

        for i in range(nr):
            for c in self.classes_:
                if self.case_ == 1:
                    edist2 = euc2(self.means_[c], T[i])
                    prior = .5
                    prior = np.log(prior)
                    disc[c] = -edist2 / (2 * self.varavg_) + prior #np.log(self.pw_[c])
                    #print(disc[c])


                elif self.case_ == 2:
```

```
                mdist2 = mah2(self.means_[c], T[i], self.covavg_)
                disc[c] = -mdist2 / 2 + np.log(self.pw_[c])
            elif self.case_ == 3:
                mdist2 = mah2(self.means_[c], T[i], self.covs_[c])
                disc[c] = -mdist2 / 2 - np.log(np.linalg.det(self.covs_[c])) / 2 \
                          + np.log(self.pw_[c])
            else:
                print("Can only handle case numbers 1, 2, 3.")
                sys.exit(1)
        y.append(disc.argmax())

    return y
```

5.1.5 Distances.py

```
'''
Created by Kevin De Angeli and Hector D. Ortiz-Melendez
Date: 2019-12-03
'''

import numpy as np
import pdb
import math

def euc2(x, y):
    # calculate squared Euclidean distance
    # check dimension
    assert x.shape == y.shape
    diff = x - y
    return np.dot(diff, diff)

def mah2(x, y, Sigma):
    # calculate squared Mahalanobis distance
    # check dimension
    assert x.shape == y.shape and max(x.shape) == max(Sigma.shape)
    diff = x - y
    return np.dot(np.dot(diff, np.linalg.inv(Sigma)), diff)

def euclideanDistance(xTest,xTrain,length):

    distance = 0

    for x in range(length):
        distance += pow((xTest[x] - xTrain[x]),2)

#         pdb.set_trace()

```

```
        return math.sqrt(distance)
```