Digit Recognizer

Khanh Tran & Luan Pham
Department of Computer Science
University of Washington

Abstract

The problem of handwritten digit recognition has been a long an open ended problem. The problem lies within the ability to come up with an efficient algorithm that boost both time complexity and accuracy. In this project, our main focus was to improve both performance and accuracy. Particularly, we will start with the lowest performance algorithms to the highest performance one in our set of algorithms. This will help better illustrating the concept of ability to come up with an efficient algorithms that can optimize both runtime and accuracy. Our list of algorithms will be Gaussian Naïve Bayes, K-Nearest Neighbor, SGD Perceptron, Adaboost (SGD logistic regression weak learners).

1 Introduction

1.1 Description

As the world start to growth, people start to get fascinating about a machine that can learn multiple diverse tasks. We are not an exception to that list. In particular, we are interested in recognition tasks. Therefore, we choose handwritten digit recognizer as the research topic for our final project. Our main goal in this project is successfully implement a digit recognition. In addition, we also want to compare the performance of different algorithms in term of accuracy and time complexity.

1.2 Data

We decided to use data retrieved from Kaggle, an online platform for machine learning related competition. There are two separated set of data. The first set is training data. This training data consists of 44,000 data point. Each data point represents for one handwritten digit image. Each handwritten digit image is a 28 by 28 pixels form a total of 784 pixels; therefore, each data point will have 785 columns, the first columns will be digit label, the rest will represents for a 784 pixels of an handwritten digit image. Each pixel is ranged from 0 to 255 inclusive. Each label is range from 0 to 9 inclusive, which is total of 10 digits to be recognize. The second set of data is test data. This test data has similar format to train data; however, it doesn't contain image label; therefore it only have 784 columns represent the 784 pixels of each single handwritten digit image.

2 Approach and Methodology

The approach will be start to layout from lowest perform to highest performance. The Description of each algorithms will describe the approach we took, the issues that we face, and the the task that we used to solve it. Besides, the methodology we describe will go from order, lowest performance to highest performance.

2.2 Gaussian Naïve bayes

Our first approach was to implement Gaussian Naïve Bayes learning Algorithms. The approach worked as follow. Based on the training data, we built two matrix and one vector. The vector is the probability of a certain class, which is P(y). The first matrix is mean matrix, this mean matrix consist of ten rows by 784 columns. Each row represent one unique single digit that range from 0 to 9. Each column will contain the mean of a single feature from total of 784 features for that digit. For instance, the first column of digit 0 will contain the mean of every single first pixel that belong to digit 0 combined. This can be done using np.mean(matrix of one unique single digit ranged from 0 to 9). The second matrix will be standard deviation matrix, this approach is the same as above; however, instead of finding mean for each feature on each unique digit, we find standard deviation. This can be done using np.nanstd(matrix of one unique single digit ranged from 0 to 0). This mean and distance matrix are being built for the purpose of calculating the likelihood of a certain feature given a certain digit, which is represented as P(x[j]|y). To calculate P(x[j]|y), we used the normal distribution density function (gaussian), which is

$$F(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

For each data point, total of 784 features, where j range from 0 to 783. Finally, we'll be a matrix of P(y)*P(x[j]|y). The matrix will have 10 rows and 784 columns. Each row will contain P(y)*P(x[j]|y) for each digit, where y is digit. Then after we got that matrix, we'll do matrix sum row by row. Then we will have a matrix of 10x1, where index of row will represent the digit number. Finally, row with max sum log(P) will be our predicted digit (Figure 1).

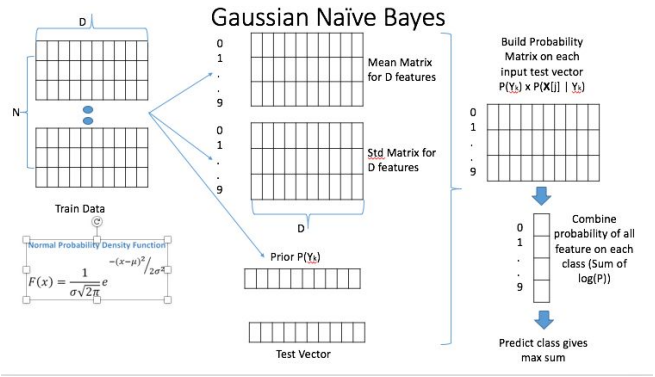


Figure 1: Gaussian Naive Bayes

2.2 K-nearest Neighbor

Even though our implementation of Naïve Bayes had extremely high runtime; however, it suffered from the accuracy performance. Therefore, our second approach was using k-nearest neighbor approach that being introduced from our lecture. However, our approach was a little different compare to provided approach from lecture. The problem that we changed because the provided approach took way too long to run when there are several difference k that we need to test on. Our approach work as following. First, We decided to build a distance matrix (figure 2) all at once for each single test data points. After that, we take one test data point, then we compute the distance between that point to one training data point. Then, we repeat the whole computing distance process on the rest of training data point on that single test data point. As the result, we'll get the vector of distance between one test data point to all training data points. Then we sort the column index base on that distance vector. After creating the distance vector for the first test data point, we'll repeat the process for the rest of data point, which we'll end up with a pre-computed distance matrix. Consequently, this distance matrix may take a little long to build, but it's worth it that we don't have to rerun the algorithm of loop and shift queue every time a new query point came in. We can simply use the distance matrix that we pre-compute and use it for all the test data points. However, this approach assumed that

we're in batch condition, where all data points are available at start of training time.

Then from our distance matrix, with sorted index based on distance.We then pick the first row with first k columns with lowest distance, which will be k-nearest neighbor for our first test data point. From that vector, we'll able to get the index that map to training labels. From that k training label, we'll do majority vote to come up with the predict y label. List majority work as following. List majority is the alternative approach beside kernel approach, which we'll consider the majority in the list as our result label. For instance, if the list is {1,2,3,1,1,1}, then our predicted label will be 1. Then repeat the same process for the rest number of rows in our sorted distance matrix, and output result of vector of predicted y. However, this approach had better overall prediction performance compared to kernel; however, we faced another problem, which is time complexity. Therefore, we decided to come up with a new way to improved the time complexity performance for our digit recognizer. According to our previous approach, the most time consuming part is when we accept a new single query point, we have to redo the whole process of looping and shift queue process through the rest of training data points. In the looping and shift queue process, the shift queue process took the longest time because it acts like a normal sorting, where it has to loop through all the current k-nearest neighbor and replace the current digit with the digit in the list where that digit had longer distance to query point compare to current digit. Subsequently, it took a tremendous amount of time to run over a large test data set. Even though, the accuracy was approximately ninety sixth percent but not feasible in real world application due to its runtime.

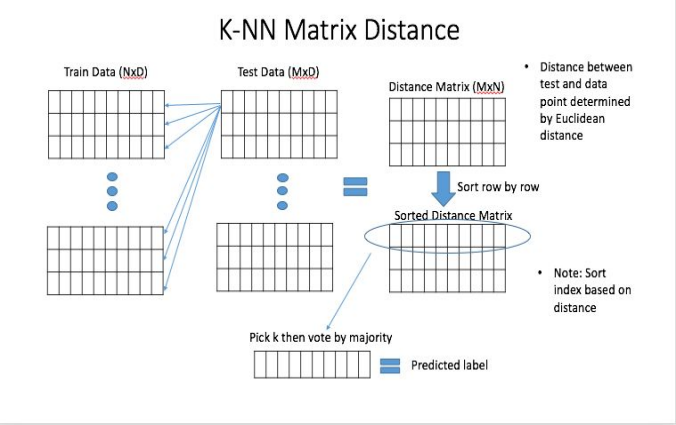


Figure 2: Matrix Distance approach

2.3 SGD Perceptron with various kernel (Online Learning)

After discussing about multiple batch learning approach that we discussed previously, we want to implement an online learning approach, which we got inspired by our previous homework 3. So far, what we have learned in lecture is all about binary classification. There is no tool to solve multiclass classification problem directly. In order to recognize all ten distinct digits, we need a strategy to transform multiclass classification to binary classification. One vs One reduction is the technique we decide to implement (Figure 3).

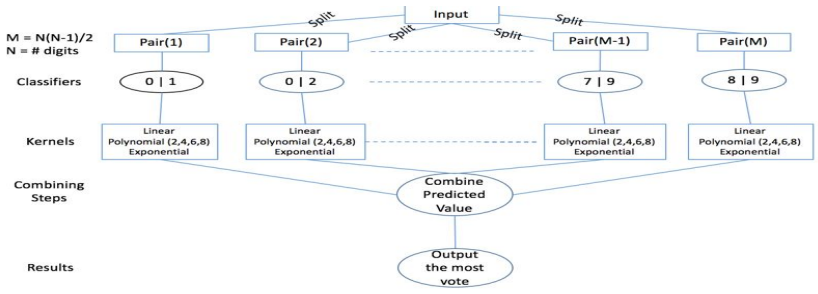


Figure 3: A step-by-step diagram of one vs one perceptron algorithm

There are ten distinct digits (45 pairs of number), we will need a total of 45 binary classifiers to vote for outcome. From given input data, we first preprocess the data into 45 binary classification subtasks. On each binary classifier, we implement perceptron algorithm with three different kernel (linear, polynomial and exponential). The final result is the majority number voted by each individual classifier.

2.4 Adaboost with Logistic Regression as weak learners

In the previous experiment with perceptron algorithm, we are able to achieve a good result with linear kernel. In this experiment, we want to apply the idea of boosting to see if we can improve both accuracy and runtime. The preprocessing steps for this approach is similar to what we do in the in perceptron algorithm; except, there are two additional steps. First, we need split input on each binary classifier into a fixed number of weak learners. Having a portion of input data on each weak learner, we apply SGD logistic regression to train these weak learners. We then apply Adaboost algorithm to create ensemble model with weighted sum of learn classifiers. In the final step, each binary classifier makes prediction based on the output of ensemble model. We take majority vote as final result (Figure 4).

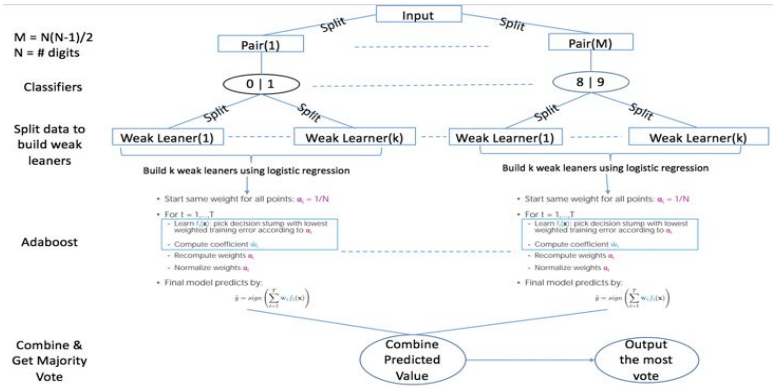


Figure 4: A step-by-step diagram of one vs one Adaboost with SGD logistic regression weak learner

3 Result and Analysis

3.1 Gaussian Naïve Bayes Summary and Analysis

From our first approach, it's clearly that naïve bayes approach is the worst approach to predict digits because the mistake error reach up to over 39%. Even though the algorithm is fast compare to other algorithms that we implemented; but the mistake rate illustrated that bayes theorem shouldn't be used to predict digits. The reason may probably because there is not enough features for it to learn; consequently, naïve bayes should work well with data that have large number of features.

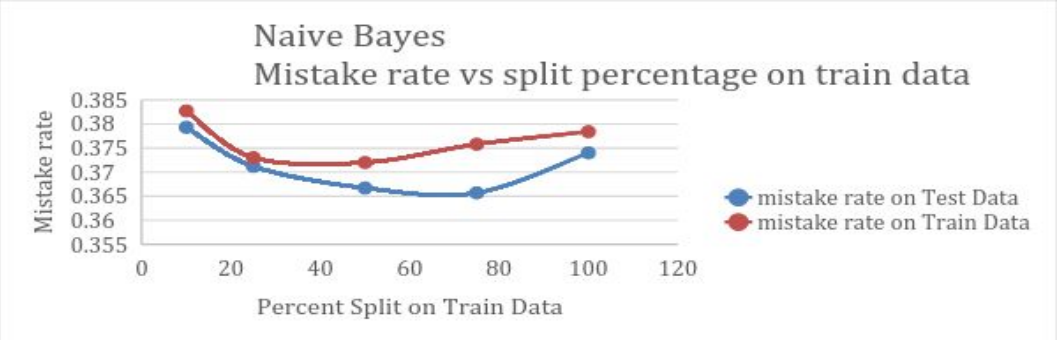


Figure 5: Mistake rate over split percentage of data

From figure 3, the train data have higher mistake rate is understandable because there are more train data point (42000) compare to test data points (22000). Therefore, when we test on 42000 points, it'll highly produce more error compare to when we test on 22000 points, with the same amount of train data. Additionally, the two curves illustrated that the the mistake rate on test set and validation test set have similar behavior over a split percentage of train data. This is understandable because we built the mean and standard deviation matrix based on our given train data. Subsequently, the mistake of predicting the wrong likelihood of a certain feature given a digit will be quite similar. The mistake rate seems to went down when we increase the amount of learn data; however, mistake rate went back up after the percent of train data increase to more than 80%. This issue may be due to the fact that our train data is not equality distributed. For instance, after training on over 80% train data, our learned data may contain more digit 1 than digit 9. In particular, this may lead to skew data to our built mean and standard deviation. Therefore, new test input doesn't always give out the expected result.

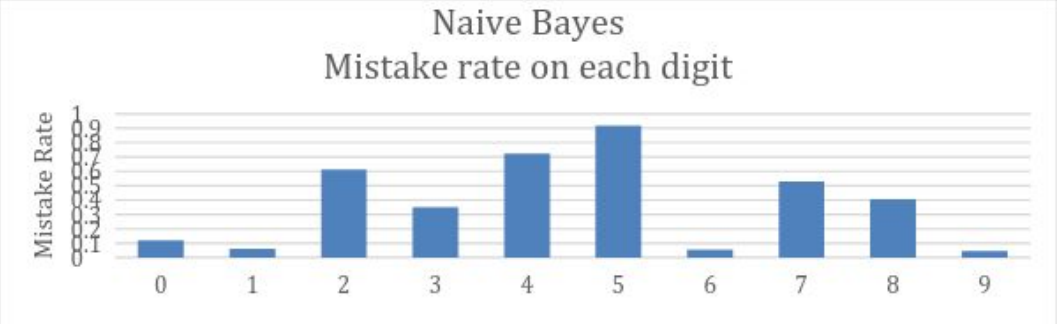


Figure 6: Mistake Rate over each digit

From figure 4, we saw that the highest misclassified digits are 2, 4, 5, 7, where 5 is the digit with highest misclassified percentage. Compare to others algorithms, the mistake rate on each digit is a little off. It predicted number 9 perfectly, whereas other algorithms have really high mistake rate on predicting this digit. Other algorithms predicted digit 4 extremely well while naïve bayes predicted digit 4 very bad. I especially mention this two digit because it doesn't make much common sense the common sense. From our perspective, 4 is very distinct shape compare to others digit such as 3, 5, 8, 9. We expected that 3, 5, 8, 9, 7 will have high mistake rate because they somehow have correlate pixel shape. But this algorithms seems to not match our common sense expectation.

3.2 K-Nearest Neighbor Summary and Analysis

From failure of Gaussian Naïve Bayes, we implemented k-Nearest Neighbor approach. The graph have clearly illustrated an extreme improvement of prediction accuracy. From the graph, k-NN have nearly 95% accuracy prediction rate compared to only 60% accuracy from Naïve Bayes. However, we still face another problem that this algorithms took extremely long time to run; even

though, we improved it with distance matrix.

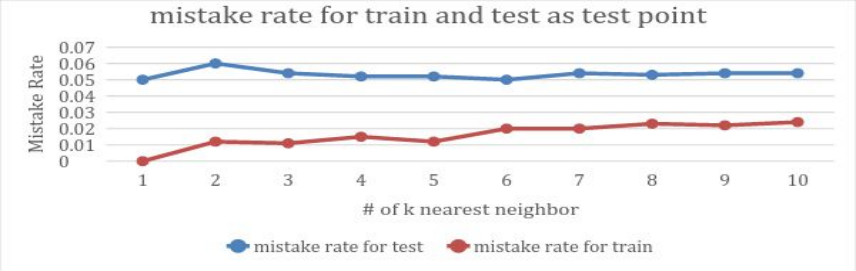


Figure 7: mistake rate for validation set and test set

From figure 5, we noticed that both curve have similar behavior over the number of k nearest neighbor that we pick. Besides, k=1 is our best parameter to gain the highest performance on accuracy. As k start to increase, the mistake rate also increase. Oppose to Naïve Bayes, our mistake rate over validation test set are overall lower compare to mistake rate over test set. It makes sense because we built our distance matrix based mainly on training data set.

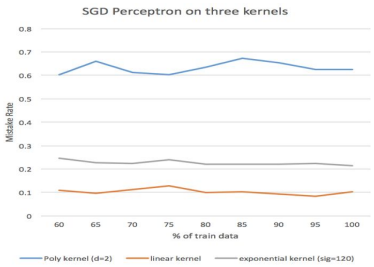
Figure 8: digit contribute to misclassified digit 8

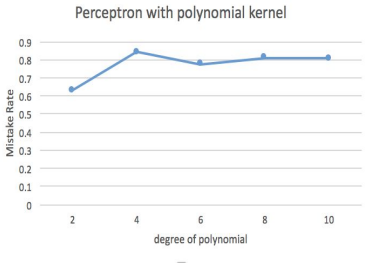
From multiple graph in figure 6, we see that the following digits have highest misclassified rate, which is 3, 5, 7, 8, 9. This matched our initial expectation perfectly. Beside, we can see that from both test and validation set, number 8 have highest misclassification rate. Beside, the following digit have highest contribution to misclassified digit 8 are digit 3,0, 5, 9. This make sense, if we take a closer look at that three digits, we see that the digit just need one and a half mark to make digit 8, whereas other digits, such as 1, 2, 4, and 7 are required much more than a mark, it required mark's shape oriented too.

3.3 Perceptron on three kernels Summary and Analysis

Figure 9a below shows the performance curves of three types of kernel. Among the three, polynomial kernel gave very high mistake rate compared to other two (Figure 9b). Therefore, we

narrow down experiments to center around linear and exponential kernel.

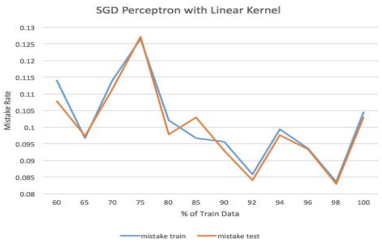


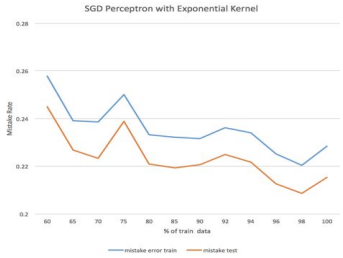


a) b)

Figure 9: a) Mistake rate on three different kernels vs train data percentage
b) Mistake rate for different degree of polynomial kernel vs train data percentage

From Figure 10 below, mistake rate's behavior on train, test data are quite similar for both linear and exponential kernel. This behavior is expected since perceptron is an online learning algorithm. It slowly aligns decision boundaries on the classifiers when there is a mistake. Linear kernel and exponential kernel are overfitting when we train them over 98% of available data set. However, linear kernel gives us a much lower mistake rate compared to exponential kernel (0.083 compared to 0.21)



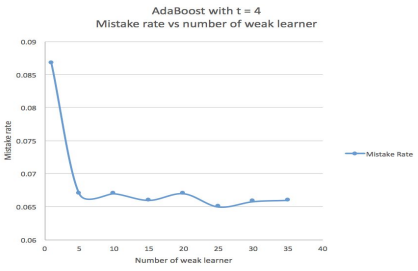


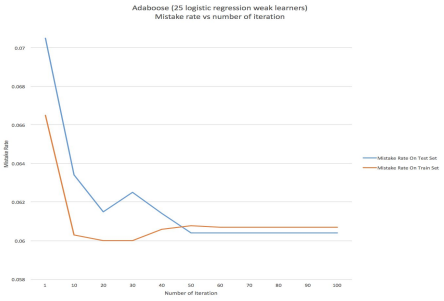
a) b)

Figure 10: a) Mistake rate on train and test set of SGD perceptron with linear kernel
b) Mistake rate on train and test set of SGD perceptron with exponential kernel

3.4 Adaboost Summary and Analysis

The implementation part of Adaboost is quite straightforward. However, there are two parameters that are extremely important in this algorithm: number of weak learner and number of iteration. It is very hard to figure out theses parameters using cross validation since our computers don't have enough computing power to run those tests. Therefore, our group decide to break experiments in smaller parts so that both of us can work on different configurations at the same time. We then compare the outcome and choose the best configuration to develop further tests.





a) b)

Figure 11: a) Mistake rate vs number of weak learner
b) Mistake rate vs number of iteration

From figure 11a above, we see that as the number of classifier increases the mistake rate start to fluctuate; eventually, it reaches the optimum when number of weak learner = 25. Having the right number of weaker we further perform experiment to figure the number of iteration for each ensemble model. Figure 11b shows the mistake curves of both train and test data. Both curves fluctuate, decreases and approaches the constant as the iterations increases. From the above graph, we figure out the optimize iteration t for train data is around 28 and t for test data is around 48 .

4 Conclusion

Final project gives us opportunities to experience different phrases of a Machine Learning project (planning, executing, testing and representing). In different phrases, we learn new set of skill, the tradeoff between different approaches. Most importantly, we are able to apply, combine theoretical concepts in lecture to tackle a completely new task, in this case building a multi class classification problem from multiple binary classifiers.

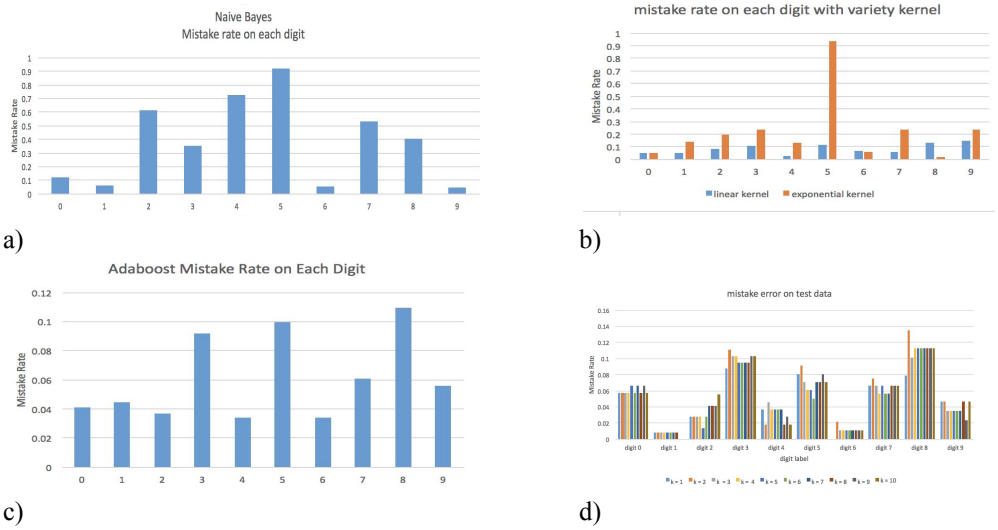


Figure 12: a) Naive Bayes accuracy b) Perceptron (linear & exponential kernel) accuracy
c) Adaboost (logistic regression weak learner) accuracy d) KNN accuracy

In summary, there is no perfect algorithms; each has some advantages over others and each performs recognition task on certain digits better than others (Figure 12 shows the accuracy result for each individual digits). KNN gives very high accurate result (96% accuracy) but slow in run-time (up to six hours). On the other hand, Naive Bayes really fast in run-time but low in accuracy (63% accuracy on 28000 test data in 20s). Adaboost and Perceptron are something in between. They both gives us medium accuracy with reasonable run-time. Figure 12 shows us the three digits that have high mistake rate error: 3, 5 and 8.