Pattern Recognition Homework (2)

OvO, OvA, Softmax & Naïve bayes

Instructed by Dr. ..

Abbas Mehrbanian

abbas.mrbn@gmail.com, Stu ID: 40130935

Reza Tahmasebi

saeed77t@gmail.com, Stu ID: 40160957

Table of Contents

1 One Vs. One

The one-vs-one model is a superb heuristic technique that uses the binary classification algorithm to classify multi-class datasets, much like the one-vs-all model. Additionally, it divides multi-class datasets into issues with binary classification. The one-vs-one classification model separates datasets into one data file for every class versus every other class, in contrast to the one-vs-rest approach, which divides datasets into a single binary assembly of data for each category.

This method has to call binary classifier $\frac{N(N-1)}{2}$ (N is a number of classes) times. For example, if we have three classes, we have to compare class one and class two together and then compare class one and class three together and then class two and class there together to train our model.

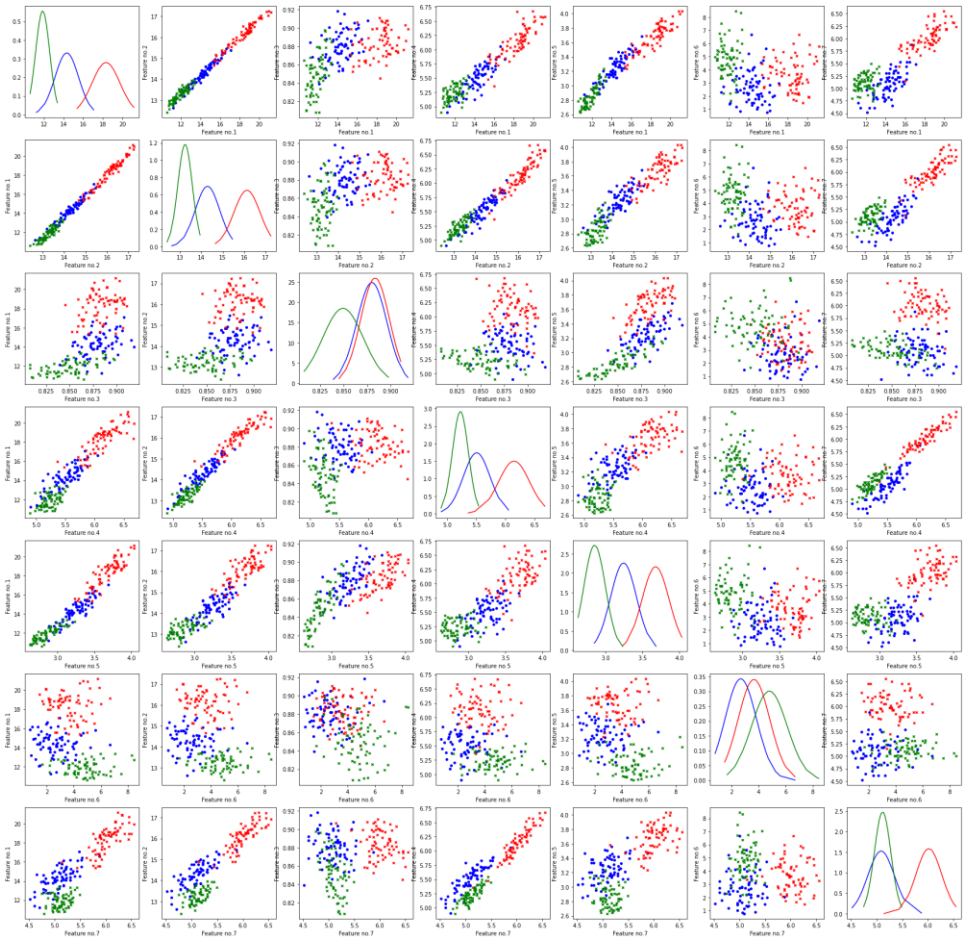

Figure 1: all dimensions of data plotted in 2 by 2 charts

1.1 Implementation

First, the dataset has been read with the *genfromtxt* method inside the numpy library. After that, data was split by class numbers and broken to test and train data. Then bias 1 was added to train and test data with *hstack*. Then with the *cal_normal_dist* function, data is plotted two by two dimensions which can be seen in Figure 1.

1.1.1 *sigmoid* function

Computes and returns the value of sigmoid function with the following formula:

$$sigmoid = \frac{1}{1 + e^{-x}}$$

1.1.2 *compute_loss* function

This function gets actual labels and predicted labels as arguments, and it calculates the cost of the algorithm.

1.1.3 *one_vs_one_data* function

This function gets features matrix and labels and also two label numbers i and j as its inputs and modifies the given data in way which only two classes of data remain and then returns the modified data. To do such task we used *logical_or* function from *numpy* library to choose features data with respect to given labels from labels vector.

1.1.4 *binary_classifier_model* function

This function gets x value, y value, alpha number, weight array and the iterations number. It calculated the parameters(weights) value with the following formula:

$$\theta_i = w_i - \alpha \left(\frac{1}{m}\sum_{i=0}^{m}\left(h_\theta\left(x^{(i)}\right) - y\right)x^{(i)}\right)$$

We also calculate the loss and adds them to an array per iteration in this function with the following formula so that we can plot cost/iterations chart later.:

$$J(y,\hat{y}) = \frac{-1}{m}\left(\sum_{i=1}^{m} y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right)$$

And finally, this function returns two values: loss history and weights.

1.1.5 *get_accuracy* function

This function gets features matrix, labels vector and weights as arguments, and calculates the accuracy of given model and returns the accuracy number.

1.2 Results

Since the number of calls to binary classifier in One vs. one approach is $\frac{N(N-1)}{2}$ where N is the number of classes, we used a nested loop to have all possible models which in our case there total of 3 classes and therefore 3 models (0vs1, 0vs2, 1vs2). Inside this loop we used the *one_vs_one_data* function to get modified data suitable for Ove vs. one algorithm, and then we called *binary_classifier_model* to train our model and finally calculate weights and loss history.

After that, we plotted the cost function/interactions chart for this algorithm.

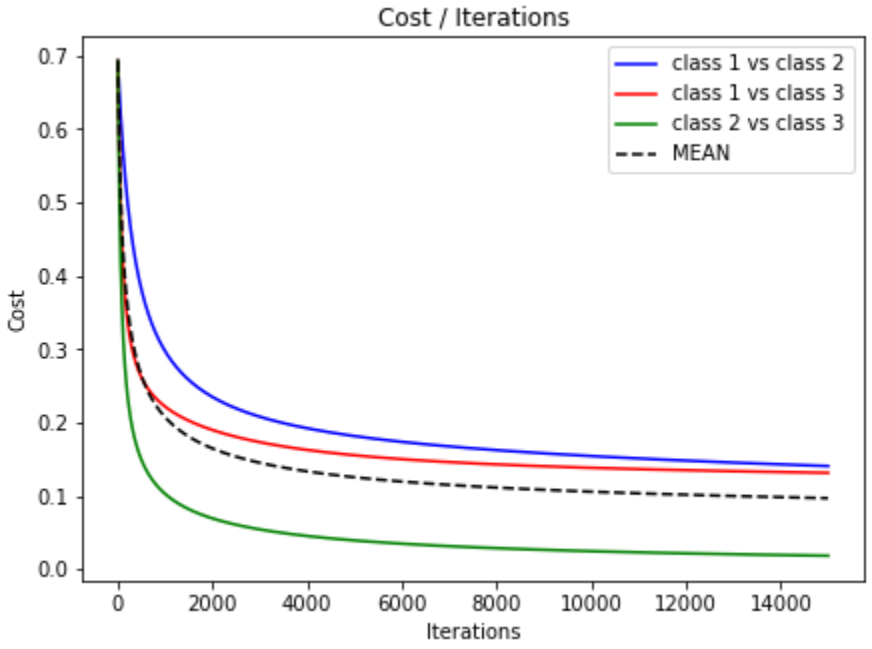

Figure 2: plot of the cost function/iterations for one vs. one algorithm

To calculate accuracy, we didn't use get_accuracy function which we discussed above. Since we use *argmax* function from *numpy* library, this approach caused some issues like ignoring one of classes in computing the prediction values. To solve this issue, we

calculated our predictions using *predict_labels* function which accepts features matrix and calculated weights as its inputs and calculate predictions for every 3 models possible and finally assign the unknown data to class based on maximum number of times that the data point was assigned a class.

This approach of calculating accuracy gave us the results below:

Training accuracy	94.55%
Test accuracy	90.47%

It can be seen from Figure 2 that the convergence iteration for the One vs. One algorithm is around the 11000^{th} iteration.

2 One Vs. All

Let me first give a quick explanation of the one-vs.-all classification principles. Consider a classification problem where there are N different classes. In this case, we'll have to train a multi-class classifier instead of a binary one. One-vs-all classification is a method that involves training N distinct binary classifiers, each designed for recognizing a particular class. Then those N classifiers are collectively used for train the model.

2.1 Implementation

There are two types of implementations available in this code; the difference is that in one method we pre-process our data in such a way that training labels changes into an (m, 3) matrix, and in the other approach we call binary classifier 3 times to show that binary classifier gets called 3 times in total.

2.1.1 *one_vs_all_data* function

This function gets labels matrix and 1 class value and modifies the labels in such a way that the given class has the value of 1 and all other classes take the value of 0 in labels matrix.

2.1.2 Method 1

In this method, first, we create an array with the shape of (m, 3) where m is the size of our samples, then, in the new labels matrix for each column representing a class, we put 1 where in the rows belonging to that class.

After that, we call the *binary_classifier_model* function to calculate weights and loss history.

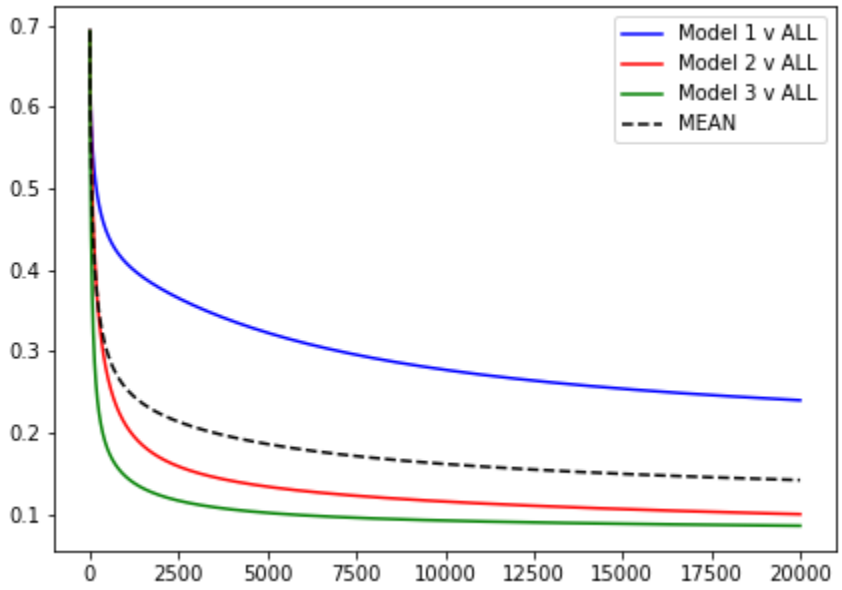

Figure 3: plot of the cost function/iterations for one vs. all algorithm method 1

Then we calculate the accuracy with the *get_accuracy* function we discussed before. The results are shown in the table below:

Training accuracy	93.87%
Test accuracy	90.47%

2.1.3 Method 2

In this method, we use a for loop in the range of the number of classes (3 in our case), and call the *one_vs_all_data* function to get the modified labels value, then use the *binary_classifier_model* function to predict the loss history and weights for every 3 models. The goal of implementing this method was to show that the number of calls to binary classifier in One vs. all approach is N, which is the number of classes (3 in our case).

After that, we plot the costs/interations using the *matplotlib* library. This plot can be seen in Figure 4.

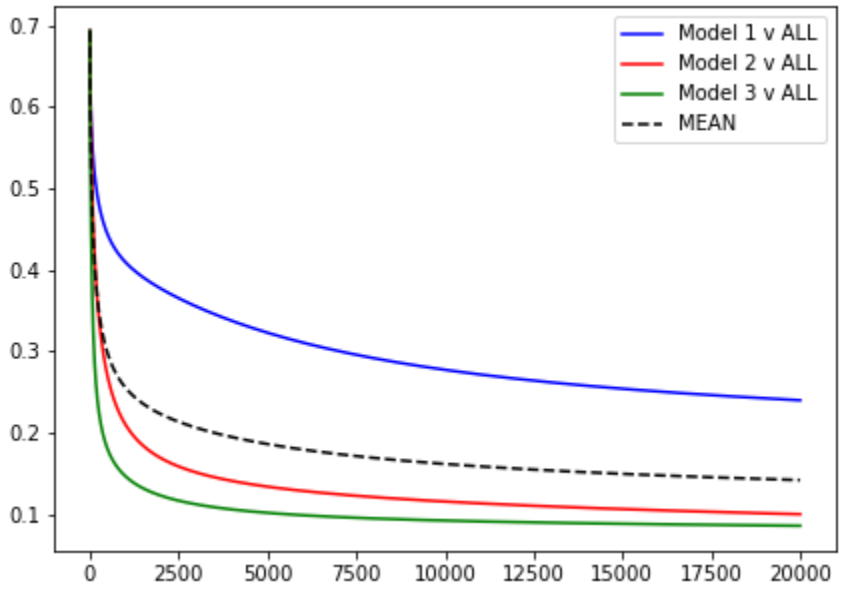

Figure 4: plot of the cost function/iterations for one vs. all algorithm method 2

After plotting the cost function, we used the *get_accuracy* function to calculate the accuracy of our model in train and test data. The results are shown in the following table.

Train data accuracy	93.87%
Test data accuracy	90.47%

It can be seen from Figure 3 and Figure 4 that the convergence iteration for One vs All algorithm in both methods is same and it's around 17500th iteration.

3 Softmax

The softmax function transforms a vector of K real numbers into a probability distribution of K possible outcomes. It is also known as softargmaxor, the normalized exponential function. It is a multidimensional generalization of the logistic function applied in multinomial logistic regression. To normalize a neural network's output to a probability distribution across projected output classes based on Luce's choice.

The softmax function takes as input a vector z of K real numbers. It normalizes it to a probability distribution consisting of K probabilities proportional to the exponentials of the input values. This means some vector components may be harmful or greater than one before applying softmax. Also, the sum may be different from 1. But after applying softmax, each component is in the interval (0,1). and the components will add up to 1, so that they can be interpreted as probabilities. The larger the input component, the greater the probability.

3.1.1 *Softmax* function

This function gets z as argument and uses the following formula to calculate softmax function value:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{i=1}^{k} e^{z_i}}$$

And it returns the calculated value.

3.1.2 *softmax_train* function

This function gets features matrix, labels vector, alpha (learning rate), initial weight array, bias and the iterations number. It calculates the loss with the following formula:

$$J(y,\hat{y}) = \frac{-1}{m}\left(\sum_{i=1}^{m} y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right)$$

And also calculates the weights with the following formula:

$$\theta_i = w_i - \alpha\,(\frac{1}{m}\sum_{i=0}^{m}\left(h_\theta\left(x^{(i)}\right) - y\right)x^{(i)})$$

After all calculations, the function returns two values: loss history and weights.

3.1.3 *get_softmax_accuracy* function

this function gets features matrix, labels vector and weights as arguments, and then it calculates the accuracy of our trained model.

3.1.4 Running softmax

First, we define number of iterations, alpha rate and initial weights. Then we use the *softmax_train* function to get the weights and the loss history.

With these two values, we can plot the costs/iterations chart.

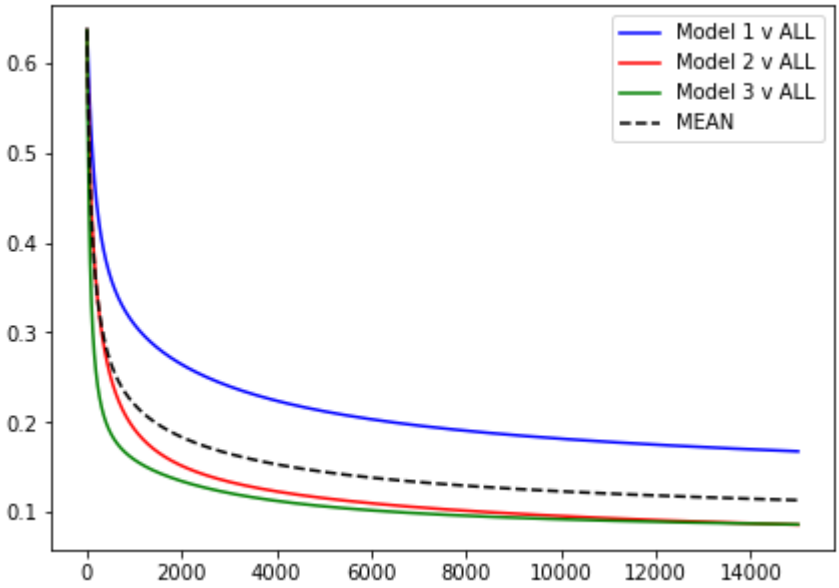

Figure 5: plot of the cost function/iterations for softmax algorithm

After plotting the costs/iterations chart, we call the *get_softmax_accuracy* function to calculate the accuracy of our model.

The results are shown in the table below.

Training accuracy	93.12%
Test accuracy	92.06%

It can be seen from Figure 5 that the convergence iteration for Softmax algorithm (is around 12000^{th}-13000^{th} iteration.

4 OvO vs OvA vs Softmax

To compare these algorithms together we have to compare their test accuracy to see how they performed in predicting unseen data.

Method	**Test accuracy**
One vs. One	90.47%
One vs. All	90.47%
Softmax	92.06%

As it can be seen Softmax performed better in predicting unseen data in most cases.

5 Naïve Bayes

5.1 Introduction

Handwriting recognition is the ability of a computer to interpret handwritten text as the characters. In this assignment we will be trying to recognize numbers from images. To accomplish this task, we will be using a Naive Bayes classifier.

Naive Bayes classifier is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. These algorithms work by combining the probabilities that an instance belongs to a class based on the value of a set of features.

For example, to understand how does it help, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as 'Naive'.

In this case we will be testing if images belong to the class of the digits 0 to 9 based on the state of the pixels in the images.

5.2 Pre-processing

The images are given as text file. Every 28x28 block in the images file is representing a digit in range 0 to 9. To read this data we wrote a function named *read_data* which accepts the text images file path as an input and returns a numpy array of size m (m = the number of samples) that its elements are 28x28 matrices of 0 and 1 representing each digit image. *read_data* reads the file line by line and replaces + and # symbols with 1 and whitespaces with 0 and adds the final matric every 28 lines, since our text digit images are 28x28 blocks, to add every number to the data array.

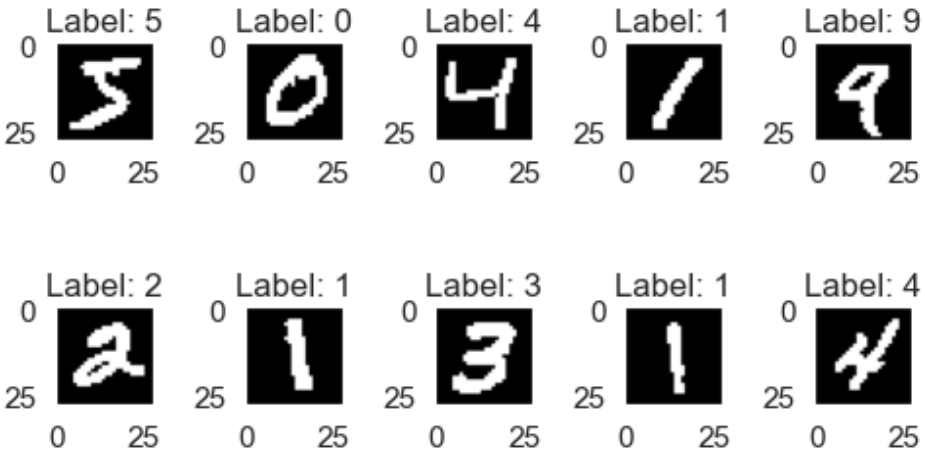

Figure 6: plotting of 10 first digit images of training images

To read labels we used *read_csv* function from *pandas* library which we worked in a same way in previous homework. We also turned them into a numpy array to be the same type as out feature matric and flatten them using *flatten* function in *Numpy* library.

We also plotted some of our samples using *imshow* function in *matplotlib.pyplot* library. The first 10 samples of training images are shown in Figure 6.

5.3 Training

In this phase, we are learning the likelihoods that a feature $F_{i,j}$ has a binary value of $f \in \{0, 1\}$ given that the image is of class $c \in \{0, 1, ..., 9\}$, which summarizes to following formula.

$$P(F_{i,j} = f \mid \text{class} = c) = \frac{\#\ of\ times\ F_{i,j} = f_{when\ class\ =\ c}}{Total\ number\ of\ training\ samples\ where\ class = c}$$

If we encounter a zero probability it'll cause issues like canceling out any non-zero probability from other features or leading to undefined values as we'll solve problem of multiply small probabilities by using logarithm later. To ensure this wouldn't happen we'll use a technique called Laplace Smoothing. Laplace smoothing is a smoothing technique that handles the problem of zero probability in Naïve Bayes. Using Laplace smoothing, we can represent $P(F_{i,j} = f \mid \text{class} = c)$ as:

$$P(F_{i,j} = f \mid \text{class} = c) = \frac{\#\ of\ times\ F_{i,j} = f_{when\ class\ =\ c} + \alpha}{Total\ number\ of\ training\ samples\ where\ class = c + \alpha * K}$$

Where α represents the smoothing parameter, K represents the number of dimensions (features) in the data. In our case since our features are binary: K = 2.

We implemented $P(F_{i,j} = \text{f} \mid \text{class} = \text{c})$ as a function named *estimate_pixel_probabilities* which accepts images(X) and labels(y) as inputs and returns a python dictionary the likelihood probabilities. To use less loops in our codes we stored these probabilities in a (1, 784) sized array instead of (28,28) array as discussed before in theory. Since we are using binary features for our image pixels, we'll only calculate the probability of pixels being 1 and later we can subtract it by 1 and use it as 0's probability. So, in this function we count the number of times j^{th} pixel is one for each class and store it in a dictionary of classes. We also count labels for every digit (class) in the training set and store it in another python dictionary. Now that we have enough data, we can compute $P(F_{i,j} = \text{f} \mid \text{class} = \text{c})$ for every class. As discussed above we used Laplace smoothing method and we implemented the $P(F_{i,j} = \text{f} \mid \text{class} = \text{c})$ equation with Laplace smoothing as a function named *laplace_smoothing* which accepts two numbers as an array, that are number of times $F_{i,j} = f_{(when\ class\ =\ c)}$ and $Total\ number\ of\ training\ samples\ where\ class = c$, and also K and α which we discussed.

We should also estimate the priors $P(class = c)$ or the probability of each class independent of features by the empirical frequencies of different classes in the training set which leads to the following equation:

$$P(\text{class} = \text{c}) = \frac{\#\ of\ training\ samples\ where\ class = c}{\#\ of\ all\ training\ samples}$$

The equation above is implemented as a function named *compute_prior* which accepts labels(y) as input and returns prior in format of a python dictionary containing probability of each class independent of features.

Now by having these values we can enter the classification of unknown data(test) phase.

5.4 Test (Classification of unknown data)

In this phase we compute the class that it is most likely to be a member of given the assumed independent probabilities for each image computed in the training stage. To

classify the unknown images using the trained model we will perform maximum a posteriori[1] classification of the test data using the trained model. In another words this method computes the posterior probability of each class for the given text image, and then classifies the image to a class that has the highest posterior probability.

As we discussed in 5.2, we can flatten each of our 28x28 images into a single dimensional vector with 784 components. This gives us an image $x \in \{0,1\}^{748}$ and $x_i \in \{0,1\}$ and this is where the "naive" part comes in. We can treat each pixel in the image of a digit as independent random variables.

Let $h(x)$ be the digit we predict given image x.

$$h(x) = argmax_j \pi_j P_j(x)$$

In the equation above we rewrote $P(x \mid \mathrm{y} = \mathrm{j})$ as $P_j(x)$ which translates to "the probability of generating image x given that label y is j" and also rewrote $P(y = j)$ as π_j.

Remember we discussed that we can compute the 0s probability based of 1s probability.

$$P_{ji}(x_i = 1) = p_{ji}$$

$$P_{ji}(x_i = 0) = 1 - p_{ji}$$

This leads us to the following equation:

$$P_{ji}(x_i) = p_{ji}^{x_i}(1 - p_{ji})^{1-x_i}, \qquad x_i \in \{0,1\}$$

Notice that when pixel values is one, $1 - x_i$ cancels out $(1 - p_{ji})$ which is the probability of 0 pixels.

Since we found both found both π_j and $P_j(x)$ in 5.3, so we can rewrite $h(x)$ equation in more concrete terms as follows:

$$P_{ji}(x_i) = \prod_{i=0}^{783} p_{ji}^{x_i}(1 - p_{ji})^{1-x_i}$$

[1] MAP

$$h(x) = argmax_j \pi_j \prod_{i=0}^{783} p_{ji}^{x_i} (1 - p_{ji})^{1-x_i}$$

using equation above might lead to "Underflow". Underflow occurs when a computer cannot represent a floating-point number beyond some specified precision. Multiplying many small decimals together (ie. probabilities), can cause this.

To avoid this problem, we will compute them using the logarithm of each probability rather than the actual value. The logarithm has the effect of taking a decimal d and outputting $\frac{1}{d}$ which produces numbers with a whole number part and a decimal part, avoiding underflow. So, the $h(x)$ will be as follows:

$$h(x) = argmax_j \log(\pi_j) + \sum_{i=0}^{738} (1 - x_i) P_{ji} + (1 - x_i) \log (1 - p_{ji})$$

We implemented $h(x)$ as a function named *predict_digit*. This function accepts an 28x28 matrix of {0,1} representing an image, priors and probabilities and computes all estimates for each class then selects the class with highest probability among *estimates* and returns that class. We made a few changes to this implementation like we changed the $(1 - x_i) P_{ji} + (1 - x_i) \log (1 - p_{ji})$ in the equation to a condition if sequence for both 0 and 1 pixels since this phrase is representing the same thing as we discussed before. This helped us gain a better performance since log multiplications made our program quite slow in first attempts of implementation.

5.5 Results

5.5.1 Confusion matrix

We computed both Numeric and percentage confusion matrix in *get_confusion_matrix* function. First, we created a 10x10 matrix, 10 is the length of our priors, then as instructed we placed actual/true values on y axis and predicted values on x axis. We used the numeric confusion matrix to compute percentage confusion matrix by simply dividing each matrix component by sum value of its column.

We plotted out confusion matrixes using *seaborn* library *heatmap* function for a cleaner output. These outputs are shown in Figure 7 and Figure 8.

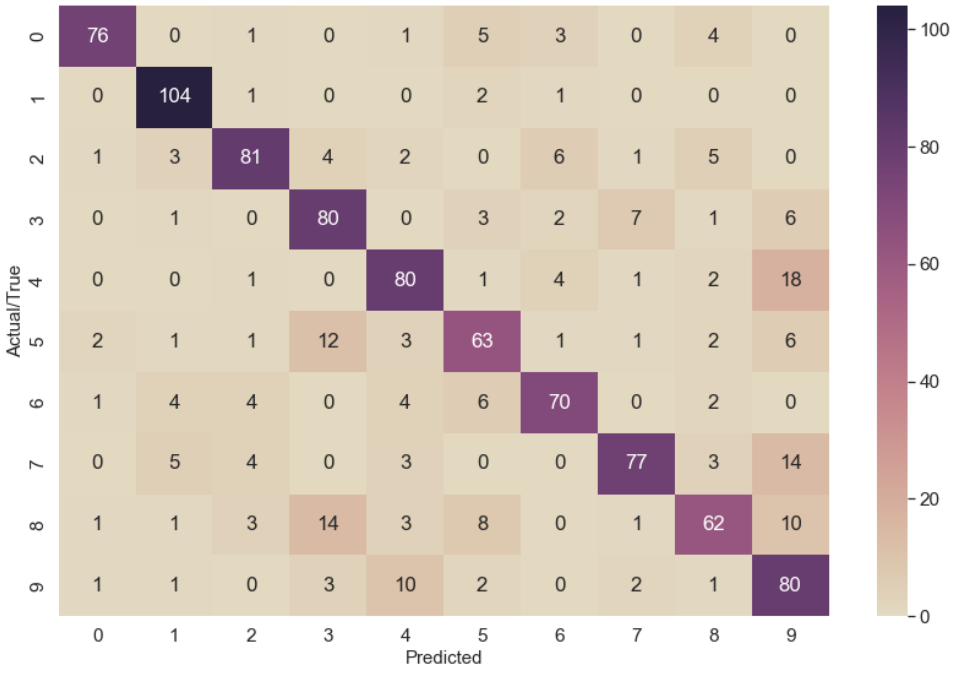

Figure 7: Confusion matrix (Numberic)

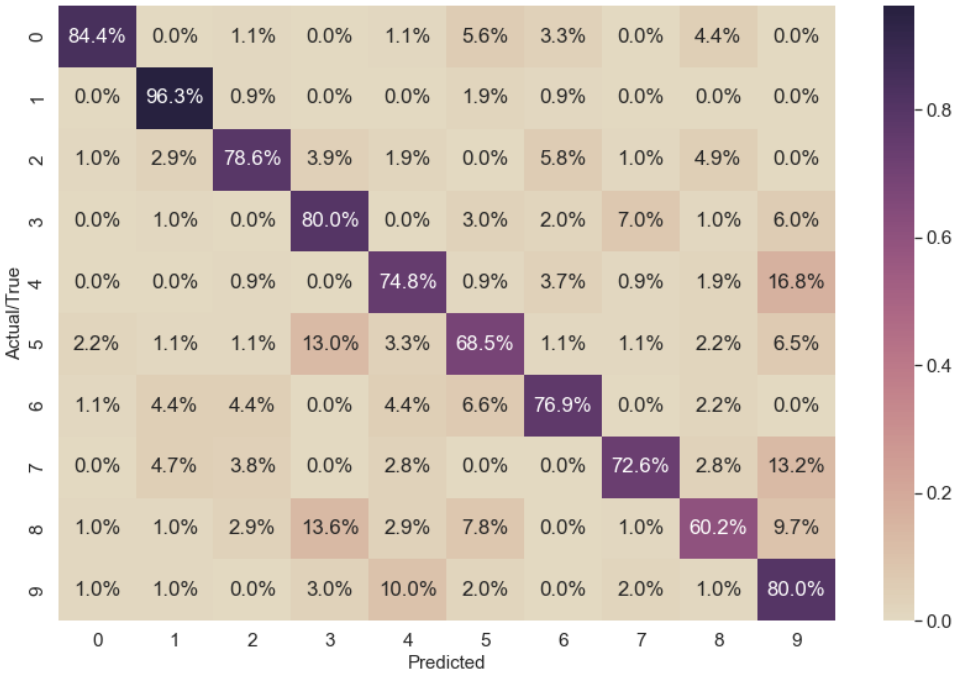

Figure 8: Confusion matrix (Percentage)

5.5.2 Accuracy

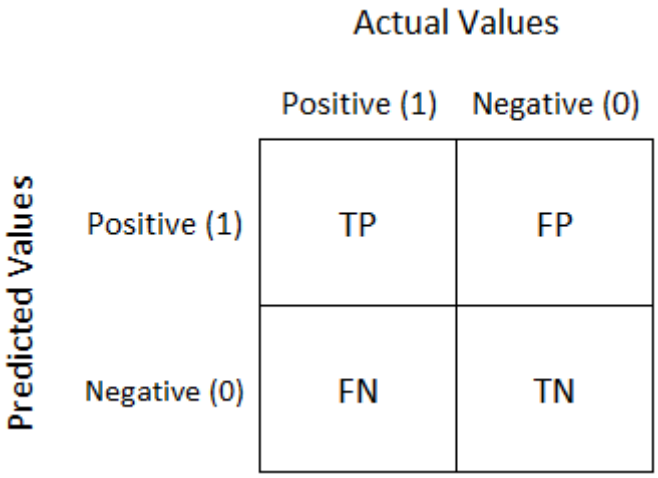

Figure 9: general confusion matrix

Accuracy can be derived from following equation with respect to Figure 9:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{sum\ of\ all\ correct\ predections\ (main\ diagonal)}{total\ number\ of\ labels}$$

This equation is implemented as a function named *cal_accuracy* which accepts confusion matrix and total number of labels as its inputs and returns calculated accuracy.

5.5.3 Precision & Recall

Precision can be derived from following equation with respect to Figure 9:

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

Recall can also be derived from following equation with respect to Figure 9:

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

Both of these equations are implemented as functions named *cal_precisions* and *cal_recalls* in order. They both accept the confusion matrix as input and return calculated values according to equations above.

5.5.4 F1 Score

F1 Score can be derived from following equation with respect to Figure 9:

$$F1Score = \frac{2TP}{2TP + FN + FP}$$

We can also drive this equation by having precision and recall values. Such that the final equation for calculating F1 score is:

$$F1Score = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}} = 2 \times \frac{Recall \times Precision}{Recall + Precision}$$

We use this equation for implement a function named *cal_f1_scores* which accepts recall and precision values and returns the f1 score value.

To report these measurements, we discussed above we wrote a general function named *report_results* that accepts confusion matrix and total number of labels as it's input and prints Recall, Precision and f1 score for every class and also prints the total accuracy of our model. The results printed by this function are provided in Table 1.

Table 1: Results table of recalls precisions, f1 scores and accuracy

Class	Recall	Precision	F1 Score
0	84.44%	92.68%	88.37%
1	96.3%	86.67%	91.23%
2	78.64%	84.38%	81.41%
3	80.0%	70.8%	75.12%
4	74.77%	75.47%	75.12%
5	68.48%	70.0%	69.23%
6	76.92%	80.46%	78.65%
7	72.64%	85.56%	78.57%
8	60.19%	75.61%	67.03%
9	80.0%	59.7%	68.38%
Accuracy		**77.3%**	