

## CSE 574: Project 3

### Recognizing Handwritten Digits from Images

11/21/2018

Mrudula Y

UB Person No: 50290843

## I. PROBLEM STATEMENT

We have handwritten images of the 10 digits in a dataset called the MNIST dataset. The images are of size  $28 \times 28$ . Our aim is to train 4 models using 4 different classifiers namely SVM, Random Forest, NNUeral Networks and Softmax Logistic Regression to predict which digit a particular image represents when new image data is given to the models. We have to analyze the performance of these 4 classifiers. Also, there is another digit image dataset called the USPS dataset. Our aim is to test our 4 classifiers on this USPS dataset and see how they perform on a completely new data.

## II. GENERAL IMPLEMENTATION DETAILS

1. Total number of features is  $28 \times 28 = 784$  as the images are scaled to size of  $28 \times 28$  and flattened to get 784 pixel values which are features. This works as all the images are centered and the pixel values represent the digits in the images well.
2. Total number of classes is 10 as there are 10 digits from 0-9. All images need to be classified as belonging one of these 10 classes.
3. **All classifier models were trained and validated on MNIST dataset.** All the classifier models were tested on MNIST and USPS dataset. **The training dataset and validation dataset mentioned in the entire report refer to MNIST data.**

## III. SOFTMAX LOGISTIC REGRESSION

### A. IMPLEMENTATION

The Softmax Logistic Regression is used for multiclass classification and has **been implemented purely using Python**. The formulas used are as per the Appendix 1 of the project 3 report. Gradient descent method was used to update weights to get the optimal weights for each class.

Details of the implementation:

1. The weights are being updated for “epoch” number of times i.e. epoch number of training features are being used starting from the beginning.
2. Weight vector has 10 columns representing the 10 classes of digits and 784 rows representing the features plus one row of bias. A bias vector is a 1 row vector of size 10 randomly initialized. The entire weight vector is initialized randomly.  
Final Dimension for weight vector :  $785 \times 10$
3. Training features vector has 785 columns representing the 784 features plus one column containing ones for incorporating the bias and 50000 rows corresponding 50000 training images which will be called data points from now on in the entire dataset.  
Final Dimension for training features:  $50000 \times 785$
4. The labels were **one hot encoded using self written functions** (refer to submitted code). In a one hot encoded matrix has 10 columns for the 10 classes and the rows equal to the number of data points. Each cell is either 0 or 1. It is 1 for the class (column number) to which the particular data point (or row) belongs to.

5. Softmax function was calculated for each row in the net input matrix which is the product of training features matrix (50000 x 785) and weight vector (785 x 10) yielding a 50000 x 10 vector. **The softmax function was self written** (refer to submitted code). The number of rows in the training features matrix varies according to the number of epochs as it is the same as the number of epochs.
6. The cost function was cross entropy. **The cost derivative was calculated using self written functions**(refer to submitted code).
7. **A small code for finding the predicted classes for the test data was self written.** This array was later used for combining the results of the 4 classifiers.

## B. HYPER-PARAMETER TUNING

List of hyper-parameters for Softmax Logistic Regression:

1. Learning Rate – rate at which the model learns i.e. step size for updating weights.
2. Lambda - the factor at which the model “unlearns” i.e. ignores outliers for a more generic hypothesis.
3. Number of Epochs - explained in Implementation section above.

## GRAPHS AND ANALYSIS

### 1) *VARYING NUMBER OF EPOCHS*

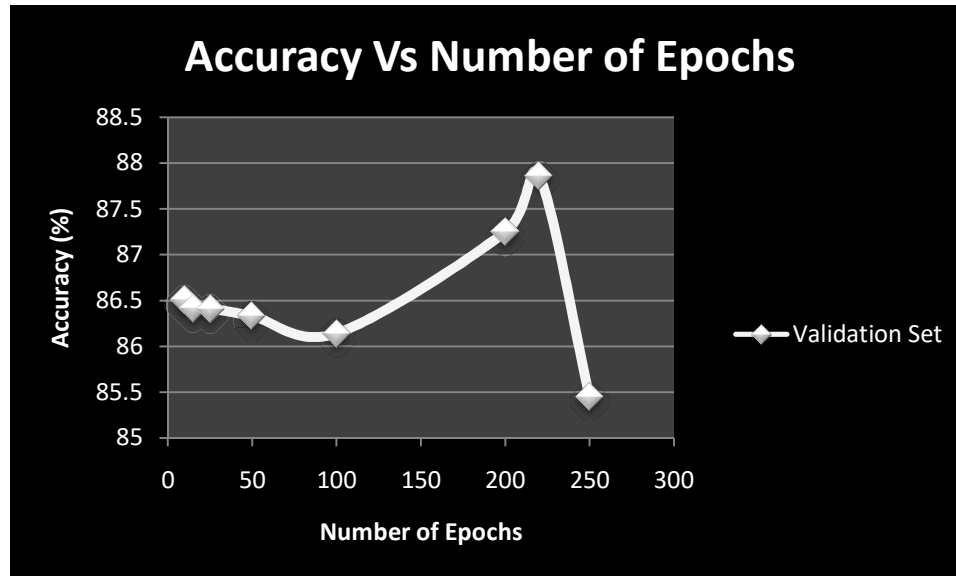
Constant hyper-parameters and their values:

1. Learning Rate: 0.01
2. Lambda: 2

Number of Epochs	Validation data set Accuracy (%)
10	86.51
15	86.42
25	86.41
50	86.33
100	86.14
200	87.25
220	87.86
250	85.45

Table 3.1 - Shows the accuracy of predictions obtained from the validation dataset when the number of epochs are varied

The Table 3.1 shows the accuracy of predictions obtained when the model was run on the validation dataset when the number of epochs for which the weights were updated are varied while keeping learning rate and lambda constant. Here the training features are taken from the beginning of the training features list. The Graph 3.1 clearly shows how the accuracy changes when number of epochs are varied.



Graph 3.1 – Shows the change in the accuracy of prediction using the validation data set when the number of epochs is varied

By observing the Graph 3.1 above, we can see that for very low values of number of epochs we see higher accuracy value but as we increase the value the accuracy starts decreasing until when number of epochs are near 100. After this point, the accuracy starts increasing and reaches its **highest at 87.86% when the number of epochs are 220**. On further increasing the number of epochs to 250 the accuracy again starts decreasing.

## 2) VARYING LEARNING RATE

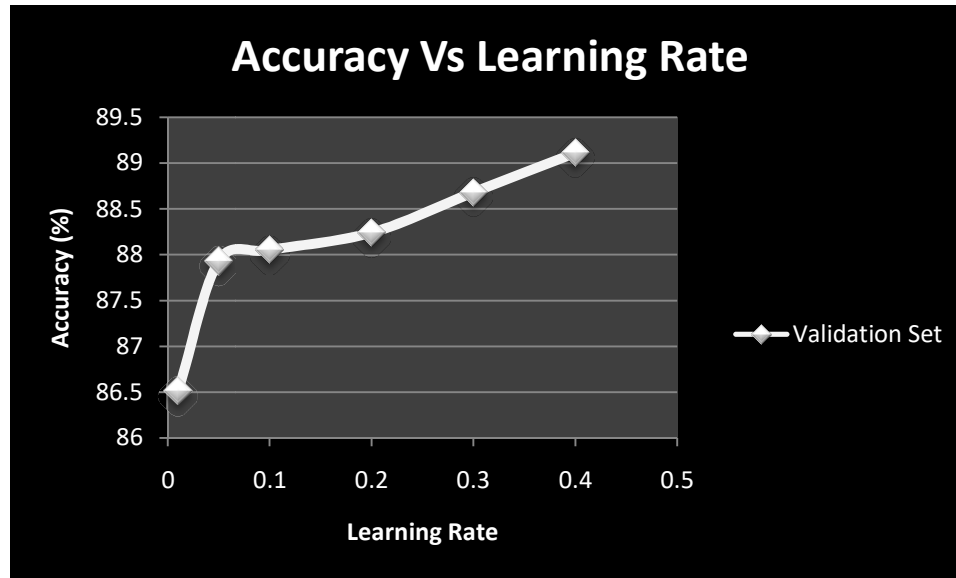
Constant hyper-parameters and their values:

1. Number of Epochs: 10
2. Lambda: 2

Learning rate	Validation data set Accuracy (%)
0.01	86.51
0.05	87.93
0.1	88.05
0.2	88.24
0.3	88.68
0.4	89.11

Table 3.2 - Shows the accuracy of predictions obtained from the validation dataset when the learning rate is varied

The Table 3.2 shows the accuracy of predictions obtained when the model was run on the validation dataset when the learning rate at which the weights were updated is varied while keeping number of epochs and lambda constant. The Graph 3.2 clearly shows how the accuracy changes when learning rate is varied.



Graph 3.2 – Shows the change in the accuracy of prediction using the validation data set when the learning rate is varied

By observing the Graph 3.2 above, we can see that as the learning rate increases from 0.01 to 0.4 the accuracy too increased and reached its highest at 0.4 learning rate. The highest accuracy obtained is 89.11%.

### 3) VARYING LAMBDA

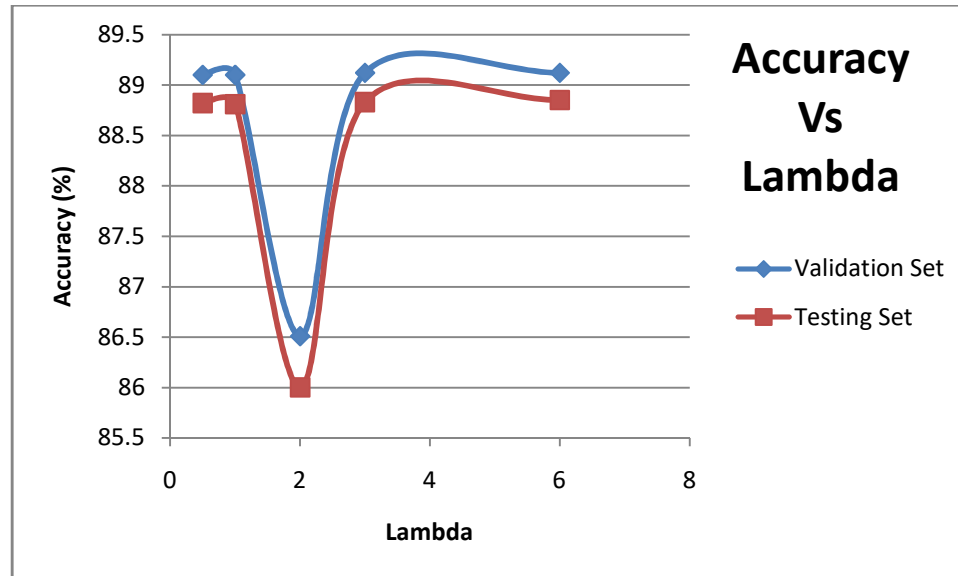
Constant hyper-parameters and their values:

3. Number of Epochs: 25
4. Learning rate: 0.01

Lambda	Validation data set Accuracy (%)	Test data set Accuracy (%)
0.5	89.1	88.82
1	89.1	88.81
2	86.51	86
3	89.12	88.83
6	89.12	88.85

Table 3.3 - Shows the accuracy of predictions obtained from the validation dataset when lambda is varied

The Table 3.3 shows the accuracy of predictions obtained when the model was run on the validation dataset when the lambda is varied while keeping number of epochs and learning rate constant. The Graph 3.3 clearly shows how the accuracy of validation and testing dataset changes when lambda is varied.



Graph 3.3 – Shows the change in the accuracy of prediction using the validation and test data set when the lambda is varied

On observing the Graph 3.3 above, we can see that as we decrease the value of lambda the accuracy increases and becomes constant after 1. From lambda value equal to 2 to 0.5 the accuracy increased by 3%. We can also see that when we increase the value of lambda above 2, the accuracy increases but it again becomes constant from 3 onwards and the value of accuracy is nearly the same which is obtained on low lambda value of 0.5. **There is slight difference in the testing set accuracy – with a higher lambda of 6 the testing gives slightly higher accuracy as compared that when lambda is low at 0.5,** though what is interesting to observe is that, the accuracy variance by order of  $10^{-2}$ .

### C. BEST RESULTS OBTAINED FOR TEST DATA OF MNIST AND USPS

#### MNIST TEST DATA

**Best Accuracy: 91.14 %**

Hyper parameter settings for the best accuracy –

1. Learning Rate: 0.4
2. Number of Epochs: 220
3. Lambda: 0.5

#### USPS TEST DATA

**Best Accuracy: 9.88 %**

Hyper parameter settings for the best accuracy –


1. Learning Rate: 0.4
2. Number of Epochs: 220
3. Lambda: 0.5

Fig. 3.1 (a) shows the screenshot of the actual code output of the best obtained for MNIST test data set and (b) shows the same for USPS data set.


**GRADIENT DESCENT SOLUTION**

In [73]: 

```
# HyperParameters
learningRate = 0.4
Lambda = 0.5
epochs = 220
```

jupyter Untitled Last Checkpoint: 34 minutes ago (autosaved)  Logout

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3

 Code

```
# calculating net input for test dataset
Ztest = numpy.dot(testingFeatures, weights)

#creating the softmax activation vector for validation dataset
softmaxT = []
count = 0
for row in Ztest:
    count = count + 1
    data = numpy.zeros((1,10))
    for i in range(numOfClasses):
        data[0][i] = softmax_function(row[i], row)
    softmaxT.append(data)
softmaxT = numpy.array(softmaxT)

# calculating accuracy for test data
matchT = 0
for row, row1 in zip(softmaxT, testingLabels):
    t = numpy.argmax(row)
    v = numpy.argmax(row1)
    if(t == v):
        matchT = matchT + 1
accuracyT = (float(matchT)*100)/mnistTestSize

print("validation set accuracy = ", accuracyV)
print("test set accuracy = ", accuracyT)

validation set accuracy = 91.38
test set accuracy = 91.14
```

Fig 3.1 (a)

## IV. SVM

### A. IMPLEMENTATION

The *scikit-learn* library was used to use the SVM classifier on the data set. SVM is one of the modules of the *scikit-learn* library which contains Support Vector Machine (SVM) algorithms.

Details of the implementation:

1. The SVC class of the SVM module was used as it can be used for multiclass classification.
2. The SVC class returns an SVM classifier object and takes arguments (hyper-parameters) such as kernel type, C and Gamma value.
3. A function called fit is used on the SVM classifier object to train the classifier the training data. It takes two parameters which are training features and training labels.
4. A function called score was used to find the accuracy the SVM classifier gives on validation and test data.
5. A function called predict was used to find the predicted classes for the test data which was later used for combining the four classifiers

### B. HYPER-PARAMETER TUNING

List of hyper-parameters for SVM:

1. Kernel – the kernel function can be specified here. Kernel functions are used to transform a non linear data space to a linear one so that a the data set can be divided using a linear plane. The *scikit-learn* SVM module provides the following kernel functions – ‘rbf’, ‘linear’, ‘poly’, ‘sigmoid’ and ‘precomputed’. The default is ‘rbf’.
2. C – This is the penalty parameter for the error term. This effects the margin size of SVM. The default is 1.
3. Gamma when kernel functions ‘rbf’, ‘poly’ and ‘sigmoid’ are used – It is the kernel coefficient when the mentioned kernel functions are used. This value intuitively signifies the amount of effect a single training data point should have on the samples which are selected as support vectors.

## GRAPHS AND ANALYSIS

### 1) **VARYING KERNEL TYPE**

Constant hyper-parameters and their values:

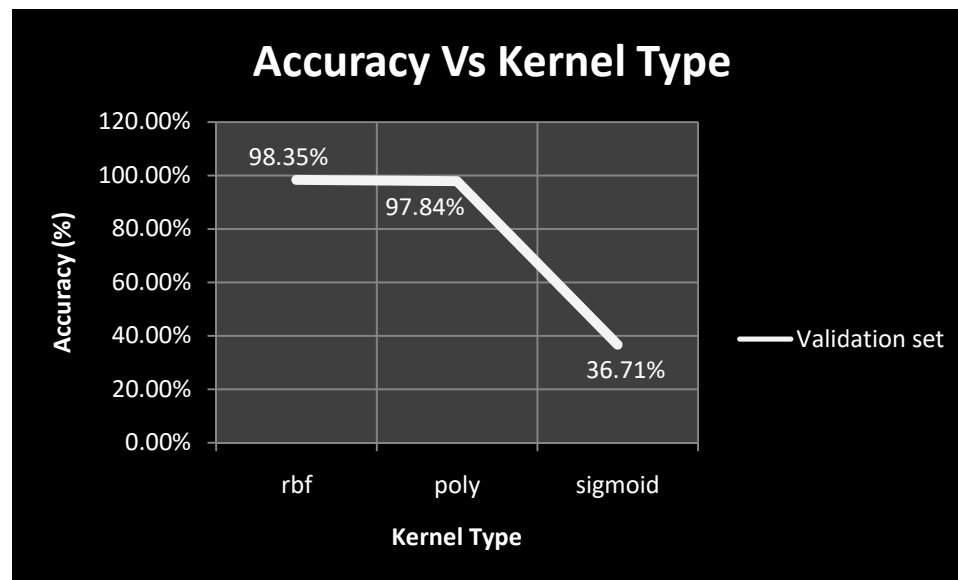
1. C : 2
2. Gamma : 0.05



Kernel	Validation data set Accuracy (%)
rbf	98.35%
poly	97.84%
sigmoid	36.71%

Table 4.1 - Shows the accuracy of predictions obtained from the validation dataset when the kernel type is varied

The Table 4.1 shows the accuracy of predictions obtained when the model was run on the validation dataset and the kernel type is varied while keeping Gamma and C constant. The Graph 4.1 clearly shows how the accuracy changes when number of epochs are varied.



Graph 4.1 – Shows the change in the accuracy of prediction using the validation data set when kernel type is varied

By observing the Graph 4.1 above, we can see that the 'rbf' kernel gives the best accuracy of 98.35%. Its worthy to note that the 'poly' kernel has also done well and it is just 0.51% less accurate that 'rbf' kernel. The 'sigmoid' kernel on the other hand has done extremely poorly with just 36.71% accuracy.

## 2) VARYING VALUE OF C

Constant hyper-parameters and their values:

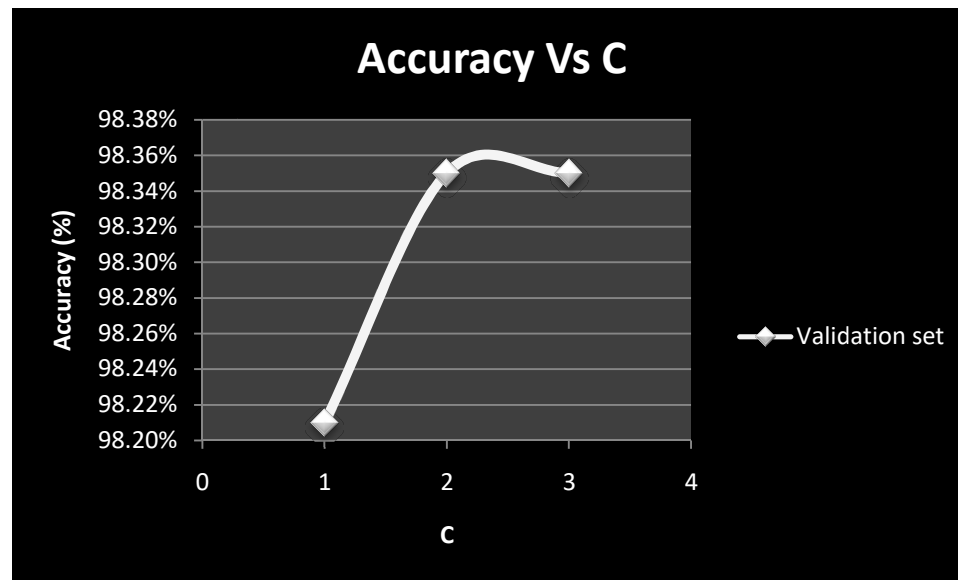
1. Kernel : 'rbf'
2. Gamma : 0.05

C	Validation data set Accuracy (%)
1	98.21%

2	98.35%
3	98.35%

Table 4.2 - Shows the accuracy of predictions obtained from the validation dataset when C is varied

The Table 4.2 shows the accuracy of predictions obtained when the model was run on the validation dataset and C is varied while keeping kernel type and Gamma constant. The Graph 4.2 clearly shows how the accuracy changes when C is varied.



Graph 4.2 – Shows the change in the accuracy of prediction using the validation data set when C is varied

By observing the Graph 4.2 above, we can see that as C value is increased the accuracy too increase but after 2 the accuracy becomes absolutely constant at 98.35% even though C is increased to 3.

### 3) VARYING GAMMA

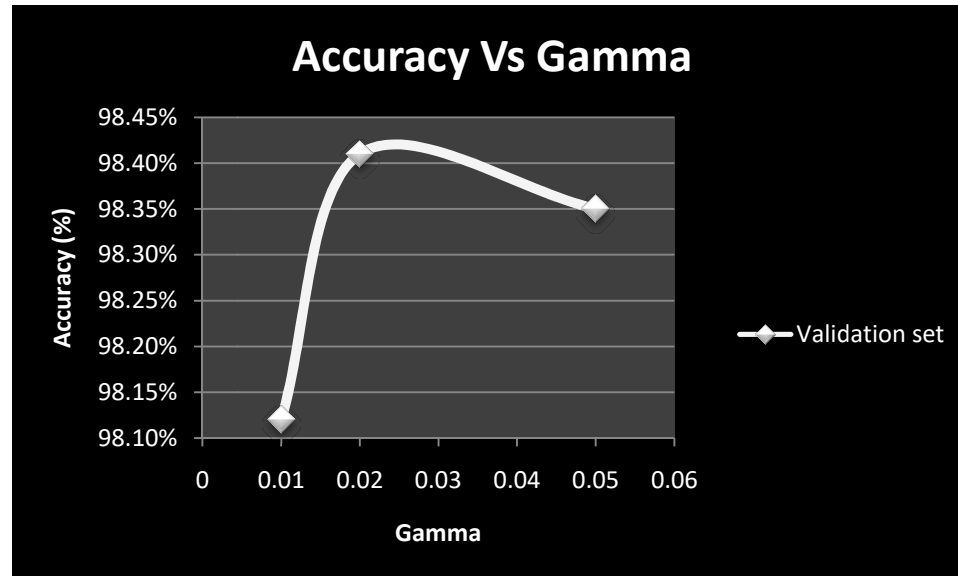
Constant hyper-parameters and their values:

1. Kernel: 'rbf'
2. C : 2

Gamma	Validation data set Accuracy (%)
0.01	98.12%
0.02	98.41%
0.05	98.35%

Table 4.3 - Shows the accuracy of predictions obtained from the validation dataset when Gamma is varied

The Table 4.3 shows the accuracy of predictions obtained when the model was run on the validation dataset and the Gamma is varied while keeping C and Kernel type constant. The Graph 4.3 clearly shows how the accuracy changes when lambda is varied.



Graph 4.3 – Shows the change in the accuracy of prediction using the validation data set when Gamma is varied

On observing the Graph 4.3 above, we can see that for a lower and higher Gamma value the accuracy decreases. The accuracy is highest at Gamma = 0.02 when the accuracy is 98.41%.

### C. BEST RESULTS OBTAINED FOR TEST DATA OF MNIST AND USPS

#### MNIST TEST DATA

**Best Accuracy: 98.35%**

Hyper parameter settings for the best accuracy –

1. Kernel : 'rbf'
2. C : 2
3. Gamma : 0.02

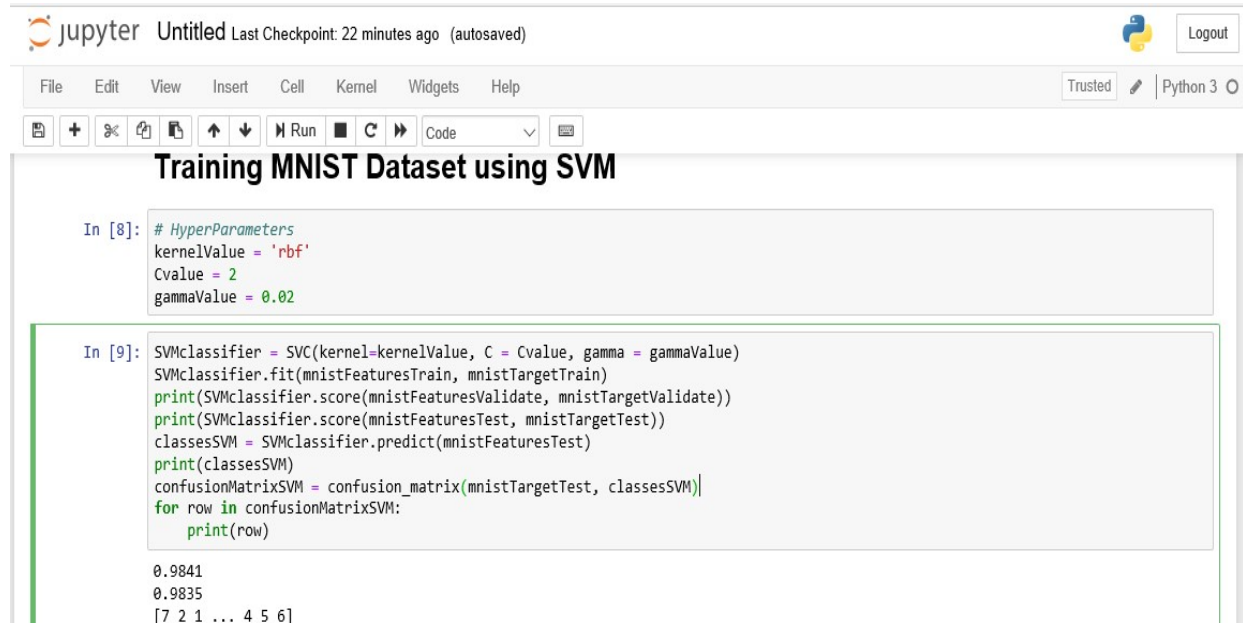
#### USPS TEST DATA

**Best Accuracy: 43.19 %**

Hyper parameter settings for the best accuracy –

1. Kernel : 'rbf'
2. C : 2
3. Gamma : 0.02

Fig. 4.1 (a) shows the screenshot of the actual code output of the best obtained for MNIST test data set and (b) shows the same for USPS data set.



The screenshot shows a Jupyter Notebook interface with the title 'Untitled' and a status bar indicating 'Last Checkpoint: 22 minutes ago (autosaved)'. The notebook has a menu bar with 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. Below the menu bar is a toolbar with icons for file operations, running, and code execution. The main area of the notebook displays two code cells. The first cell, labeled 'In [8]:', defines hyperparameters for an SVM: `kernelValue = 'rbf'`, `Cvalue = 2`, and `gammaValue = 0.02`. The second cell, labeled 'In [9]:', imports the `SVC` class from `sklearn.svm`, fits it to the training data, and prints the score on validation and test data. It also prints the predicted classes and a confusion matrix. The output of the second cell shows a score of 0.9841 on validation data and 0.9835 on test data, followed by a confusion matrix for the test data.

```

In [8]: # HyperParameters
kernelValue = 'rbf'
Cvalue = 2
gammaValue = 0.02

In [9]: SVMclassifier = SVC(kernel=kernelValue, C = Cvalue, gamma = gammaValue)
SVMclassifier.fit(mnistFeaturesTrain, mnistTargetTrain)
print(SVMclassifier.score(mnistFeaturesValidate, mnistTargetValidate))
print(SVMclassifier.score(mnistFeaturesTest, mnistTargetTest))
classesSVM = SVMclassifier.predict(mnistFeaturesTest)
print(classesSVM)
confusionMatrixSVM = confusion_matrix(mnistTargetTest, classesSVM)
for row in confusionMatrixSVM:
    print(row)

0.9841
0.9835
[7 2 1 ... 4 5 6]

```

Fig 4.1 (a)

## V. RANDOM FOREST

### A. IMPLEMENTATION

The *scikit-learn* library was used to use the Random Forest classifier on the data set. '*ensemble*' is one of the modules of the *scikit-learn* library which contains Random Forest Classifier algorithms.

Details of the implementation:

1. The RandomForestClassifier class of the '*ensemble*' module was used.
2. The object of the class RandomForestClassifier is the Random Forest classifier which will be trained further.
3. The fit function was used to train the Random Forest classifier. It takes two parameters – the training features and the training labels.
4. A function called score was used to find the accuracy the Random Forest classifier gives on validation and test data.
5. A function called predict was used to find the predicted classes for the test data which was later used for combining the four classifiers

### B. HYPER-PARAMETER TUNING

List of hyper-parameters for Random Forest :

1. Number of Trees – This basically signifies the number of randomly drawn subsets of data from the total original training data set. It is represented by the parameter  $n\_estimators$  in the RandomForestClassifier class.

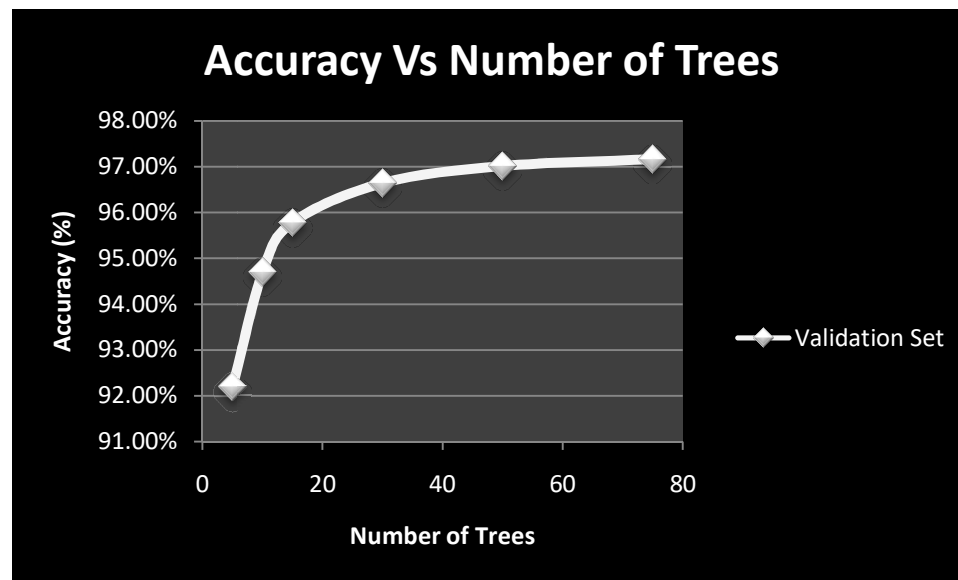
## GRAPHS AND ANALYSIS

### 1) VARYING NUMBER OF TREES

Number of Trees	Validation data set Accuracy (%)
5	92.19%
10	94.70%
15	95.78%
30	96.64%
50	97.01%
75	97.16%

Table 5.1 - Shows the accuracy of predictions obtained from the validation dataset when the number of trees are varied

The Table 5.1 shows the accuracy of predictions obtained when the model was run on the validation dataset and the number of trees are varied while keeping all other parameters constant. The Graph 5.1 clearly shows how the accuracy changes when number of trees are varied.



Graph 5.1 – Shows the change in the accuracy of prediction using the validation data set when the number of trees are varied

By observing the Graph 5.1 above, we can see that as the number of trees are increased the accuracy too increases but after 50 it kind of saturates at 97.01% and despite increasing the number of trees to 75 the accuracy doesn't change much.

## C. BEST RESULTS OBTAINED FOR TEST DATA OF MNIST AND USPS

### MNIST TEST DATA

**Best Accuracy : 97.16%**

Hyper parameter settings for the best accuracy –

1. Number of Trees : 75

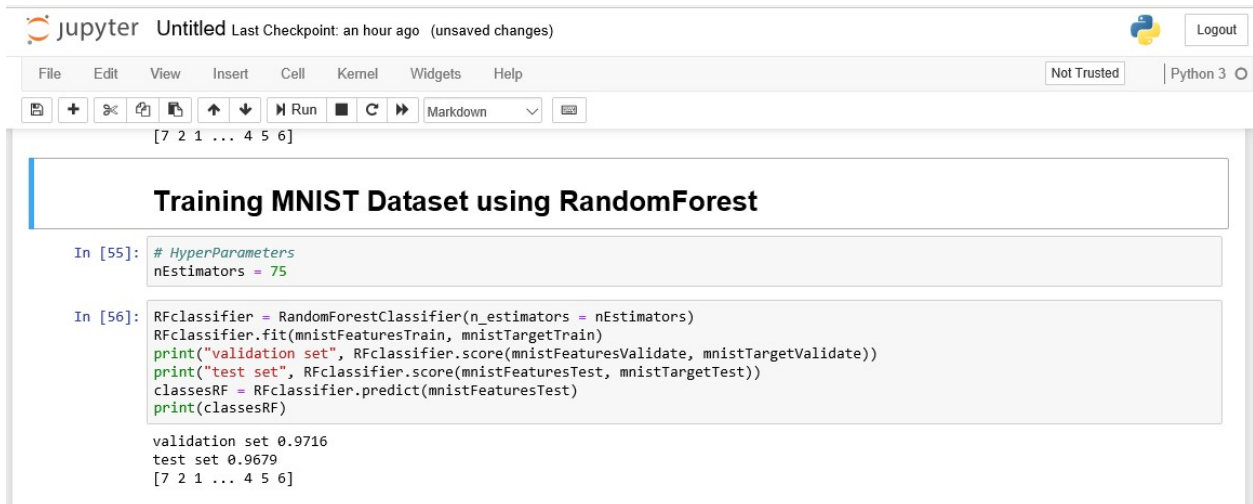
### USPS TEST DATA

**Best Accuracy: 38.63 %**

Hyper parameter settings for the best accuracy –

1. Number of Trees : 75

Fig. 5.1 (a) shows the screenshot of the actual code output of the best obtained for MNIST test data set and (b) shows the same for USPS data set.



```
jupyter Untitled Last Checkpoint: an hour ago (unsaved changes) Logout
File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3
[7 2 1 ... 4 5 6]

Training MNIST Dataset using RandomForest

In [55]: # HyperParameters
n_estimators = 75

In [56]: RFclassifier = RandomForestClassifier(n_estimators = n_estimators)
RFclassifier.fit(mnistFeaturesTrain, mnistTargetTrain)
print("validation set", RFclassifier.score(mnistFeaturesValidate, mnistTargetValidate))
print("test set", RFclassifier.score(mnistFeaturesTest, mnistTargetTest))
classesRF = RFclassifier.predict(mnistFeaturesTest)
print(classesRF)

validation set 0.9716
test set 0.9679
[7 2 1 ... 4 5 6]
```

Fig 5.1 (a)

## VI. NEURAL NETWORKS

### A. IMPLEMENTATION

The *keras* library of Python was used for the implementation of Neural Networks to train the data. The *keras* library works on the Tensorflow library. The Tensorflow library runs at the backend while using *keras*.

Details of the implementation:

1. Data loading – The function for loading MNIST data set is available in *keras* library's 'datasets' module and is called 'mnist'. The MNIST dataset is loaded using the function `load_data` which is contained in 'mnist'.
2. Partitioning – When the `load_data` function is used, it automatically returns data partitioned into training features and labels and testing features and labels – a total of 4 datasets.
3. Pre-Processing – The data returned by `load_data` function is not directly used. The training and testing features are reshaped to matrices of size 28x28. The training and testing labels are converted into binary matrices (which look similar to on-hot encoded matrix explained in the softmax logistic regression section) of size (number of data points) x 10 where 10 is the number of classes. This is done because the *keras* neural network functions require the data to be in this form to run.
4. *Keras* has a 'Sequential Model API' which basically emulates a neural network model which has linearly arranged set of layers. Using this API, we can specify customized layers and add them to our neural network set. We use this Sequential model API to define our neural network model.
5. The Sequential model API has a function called `Sequential` that returns a model.
6. On to this model, using the `add` function 2 layers were added. One is the input layer and second one is the output layer. The inputlayer for training and the output layer returns output as any one of the 10 classes.
7. The type of layer added is densely connected Neural Networks using the `Dense` function. The `Dense` function has many parameters, the parameters we specified were number of units in the layer, the activation function that needs to be applied on the input and the input size. The input size needs to be specified for the input layer which are the  $28 \times 28 = 784$  features for one data point (image). We can use same or different activation functions. The units in input layer can be varied but the number of units in output layer is fixed because the Neural Network has to give 10 outputs for the 10 classes. These three are hyper-parameters.
8. The model is compiled using an optimizer as parameter, which is also hyper-parameter. The optimizer is basically method using which the weights are updated to minimize the losses. There are many optimizers which can be specified – 'sgd' which is stochastic gradient descent, 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax' and 'Nadam'. The `compile` method also takes a loss parameter where we can specify which kind of loss we want to minimize. Here we chose cross entropy. We can also specify the metric to be used to evaluate the model. Here we chose accuracy.

9. Next, we use the fit function to train the model using the training features and training labels. Here we also mention the batch size in which the training should happen. We also specify the number of epochs the entire data should be run for training. An interesting parameter here is the **validation\_split**, which is used to specify the fraction of training set we want to be considered as validation dataset. While fitting the training data, for each epoch the validation data set is checked for accuracy which helps us visualize whether the accuracy is increasing with each epoch or else we will know there is some error in our training.
10. The evaluate function is used to find the accuracy of the test data set.
11. The predict\_classes function is used to find the predicted classes for the test data which was later used for combining the four classifiers.

## B. HYPER-PARAMETER TUNING

List of hyper-parameters for Neural Networks :

1. Number of Epochs – While training the training data using the fit function we can specify this parameter. This is the number of times/iterations the entire data set needs to be re-run by the model. The parameter is called epochs.
2. Batch Size – While training the training data using the fit function we can specify this parameter . The batch size specifies the amount of data that needs to be passed to the model at a time while training. The default is 32.
3. Optimizer – The optimizer to be used for optimizing the model to minimize the loss. There are many optimizers which can be specified – ‘sgd’ which is stochastic gradient descent, ‘RMSprop’, ‘Adagrad’, ‘Adadelta’, ‘Adam’, ‘Adamax’ and ‘Nadam’.
4. Number of units in first layer – This is specified as a parameter to the Dense function. As the name suggests it is the number of neurons/units in the layer.

## GRAPHS AND ANALYSIS

### 1) **VARYING OPTIMIZER**

Constant hyper-parameters and their values:

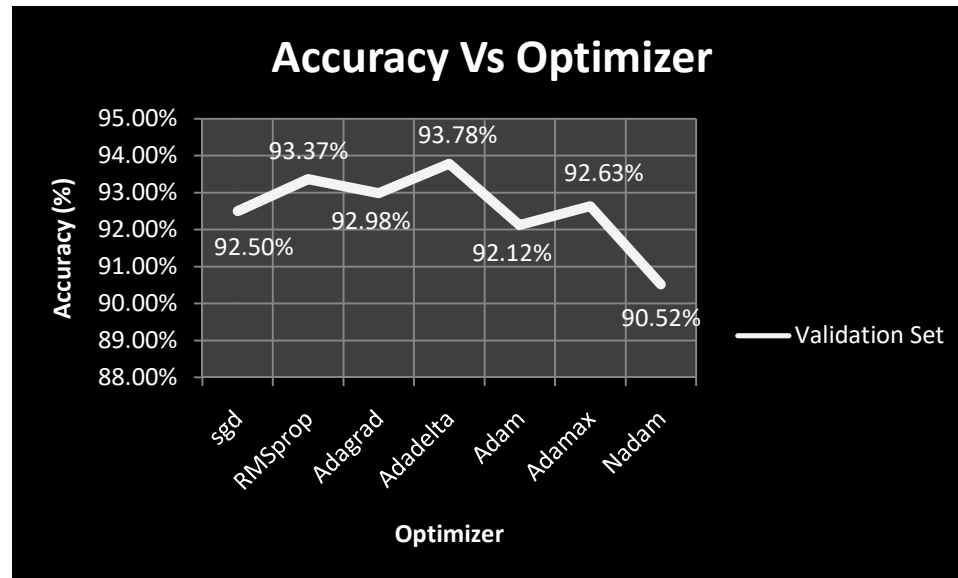
1. Batch Size : 128
2. Number of Units in Layer 1 : 32
3. Number of Epochs : 10

Optimizer	Validation data set Accuracy (%)
sgd	92.50%
RMSprop	93.37%
Adagrad	92.98%
Adadelta	93.78%
Adam	92.12%
Adamax	92.63%
Nadam	90.52%



Table 6.1 - Shows the accuracy of predictions obtained from the validation dataset when the optimizer is varied

The Table 6.1 shows the accuracy of predictions obtained when the model was run on the validation dataset when the optimizer using which the weights were updated is varied while keeping other hyper-parameters constant. The Graph 6.1 clearly shows how the accuracy changes when optimizers are varied.



Graph 6.1 – Shows the change in the accuracy of prediction using the validation data set when the optimizer is varied

By observing Graph 6.1 above, we can see that Adadelata optimizer gives the best accuracy of 93.78% and the Nadam optimizer gives the worst accuracy of 90.52%.

## 2) VARYING NUMBER OF EPOCHS

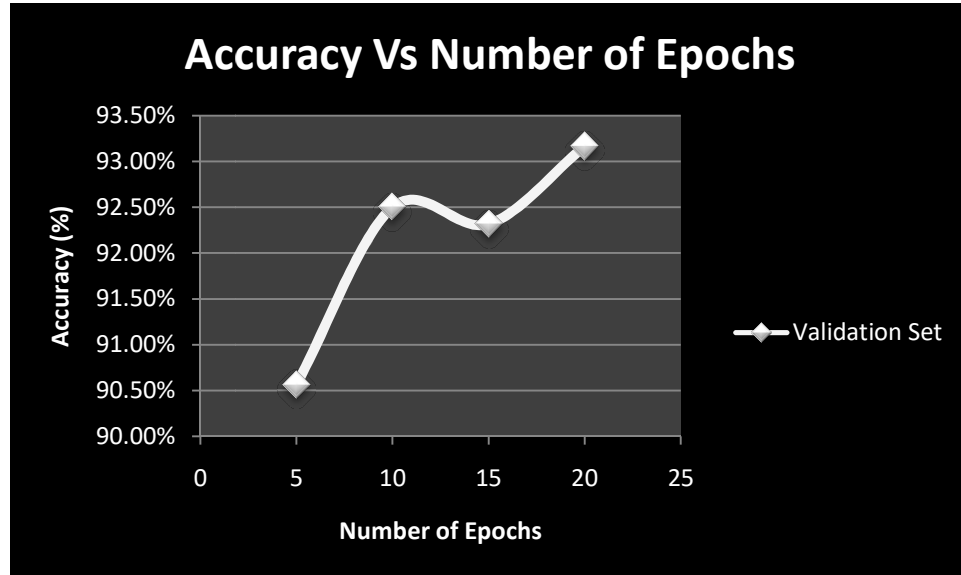
Constant hyper-parameters and their values:

1. Batch Size : 128
2. Optimizer : 'sgd'
3. Number of Units in Layer 1 : 32

Number of Epochs	Validation data set Accuracy (%)
5	90.57%
10	92.50%
15	92.32%
20	93.17%

Table 6.2 - Shows the accuracy of predictions obtained from the validation dataset when the number of epochs are varied

The Table 6.2 shows the accuracy of predictions obtained when the model was run on the validation dataset when the number of epochs for which the weights were updated are varied while keeping other hyper-parameters constant. The Graph 6.2 clearly shows how the accuracy changes when number of epochs are varied.



Graph 6.2 – Shows the change in the accuracy of prediction using the validation data set when the number of epochs are varied

By observing the Graph 6.2 above, we can see that as the number of epochs increase the accuracy also increases giving a maximum accuracy when number of epochs are 20.

### 3) VARYING BATCH SIZE

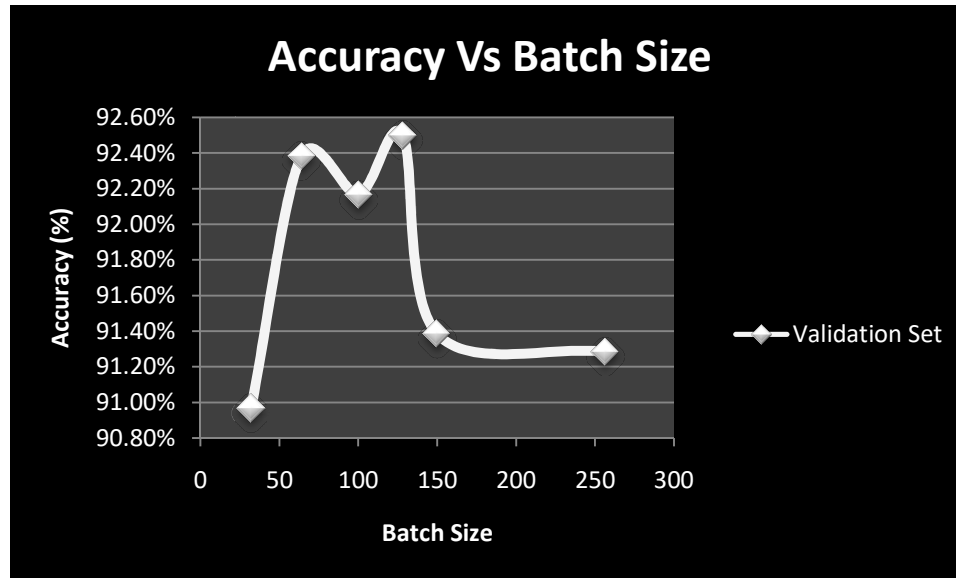
Constant hyper-parameters and their values:

1. Number of Epochs : 10
2. Optimizer : 'sgd'
3. Number of Units in Layer 1 : 32

Batch Size	Validation data set Accuracy (%)
32	90.97%
64	92.38%
100	92.17%
128	92.50%
150	91.38%
256	91.28%

Table 6.3 - Shows the accuracy of predictions obtained from the validation dataset when the Batch size is varied

The Table 6.3 shows the accuracy of predictions obtained when the model was run on the validation dataset when the batch size is varied while keeping other hyper-parameters constant. The Graph 6.3 clearly shows how the accuracy changes when batch size is varied.



Graph 6.3 – Shows the change in the accuracy of prediction using the validation data set when the Batch size is varied

By observing the Graph 6.3, we can see that there is no proper variance for the middle values between 50 to approximately 130 batch size, but we can see a very low batch size and a very high batch size are causing the accuracy to decrease. The highest accuracy is obtained at 128 batch size which is 92.50%.

#### 4) **VARYING NUMBER OF UNITS IN LAYER 1**

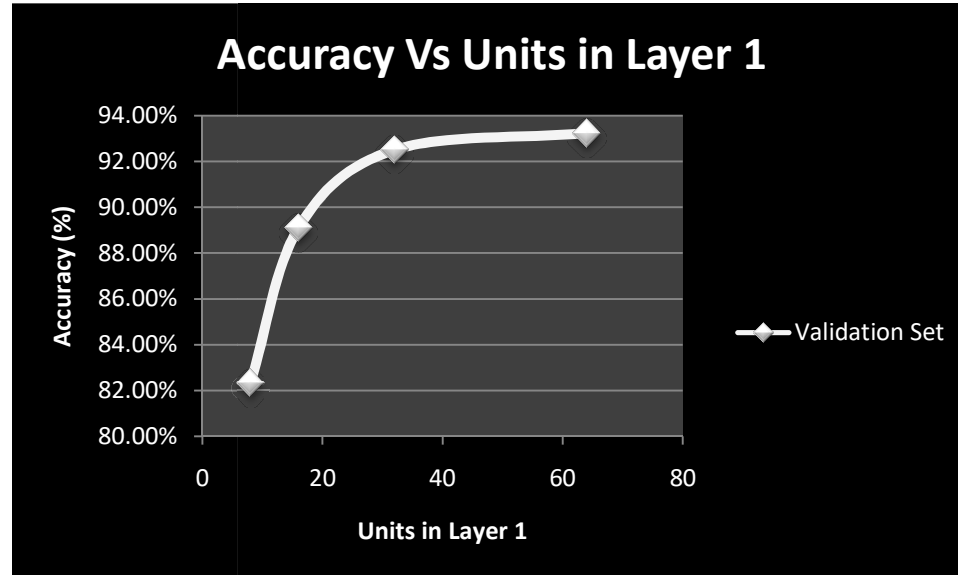
Constant hyper-parameters and their values:

4. Number of Epochs : 10
5. Optimizer : 'sgd'
6. Batch Size : 128

Number of Units in Layer 1	Validation data set Accuracy (%)
8	82.32%
16	89.10%
32	92.50%
64	93.23%

Table 6.4 - Shows the accuracy of predictions obtained from the validation dataset when the number of units in layer 1 are varied

The Table 6.4 shows the accuracy of predictions obtained when the model was run on the validation dataset when the number of units in layer 1 are varied while keeping other hyper-parameters constant. The Graph 6.4 clearly shows how the accuracy changes when number of units in layer 1 are varied.



Graph 6.4 – Shows the change in the accuracy of prediction using the validation data set when the number of units in layer 1 are varied

By observing the Graph 6.4 above, we can see that as the number of units in layer 1 increase the accuracy also increases but becomes near to constant i.e. there is no much significant change in accuracy after 32 number of units in layer 1.

## C. BEST RESULTS OBTAINED FOR TEST DATA OF MNIST AND USPS

### MNIST TEST DATA

**Best Accuracy: 94.64%**

Hyper parameter settings for the best accuracy –

1. Number of Epochs: 20
2. Optimizer : 'Adadelta'
3. Number of Units Layer 1 : 64
4. Batch Size : 128

### USPS TEST DATA

**Best Accuracy: 21.89 %**

Hyper parameter settings for the best accuracy –

- A. Number of Epochs: 20
- B. Optimizer : 'Adadelata'
- C. Number of Units Layer 1: 64
- D. Batch Size : 128

Fig. 6.1 (a) shows the screenshot of the actual code output of the best obtained for MNIST test data set and (b) shows the same for USPS data set.

```

x_train = x_train.reshape(x_train.shape[0], image_vector_size)
x_test = x_test.reshape(x_test.shape[0], image_vector_size)
y_train = keras.utils.to_categorical(y_train, numOfClasses)
y_test = keras.utils.to_categorical(y_test, numOfClasses)

In [57]: # HyperParameters
unitsLayer1 = 64
activationLayer1 = 'sigmoid'
activationLayer2 = 'softmax'
optimizerValue = 'Adadelata'
batchSize = 128
epochsValue = 20

In [58]: image_size = 784
model = Sequential()
model.add(Dense(units=unitsLayer1, activation=activationLayer1, input_shape=(image_size,)))
model.add(Dense(units=numOfClasses, activation=activationLayer2))
model.compile(optimizer=optimizerValue, loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit(x_train, y_train, batch_size=batchSize, epochs=epochsValue, verbose=False, validation_split=0.1)
print("validation set accuracy ", history.history['val_acc'])
loss, accuracy = model.evaluate(x_test, y_test, verbose=False)
print("test set loss = ", loss, "test set accuracy = ", accuracy)
classesNN = model.predict_classes(x_test, batch_size=batchSize)
print(classesNN)

validation set accuracy [0.8974999996821086, 0.9171666663487752, 0.9293333328564962, 0.9371666671435038, 0.9366666665077209,
0.9413333331743876, 0.9425000001589457, 0.9481666668256123, 0.9483333330154419, 0.9456666663487753, 0.949833333492279, 0.952166
6661898295, 0.9538333336512248, 0.9538333334922791, 0.9513333330154419, 0.9566666663487752, 0.9593333330154419, 0.9569999996821
086, 0.9606666668256124, 0.9591666663487752]
test set loss = 0.17436932053118945 test set accuracy = 0.9464
[7 2 1 ... 4 5 6]

```

Fig 6.1 (a)

## VII. COMBINING RESULTS OF FOUR CLASSIFIERS USING MAJORITY VOTING

### A. IMPLEMENTATION

The Majority Voting is nothing but taking the result which the majority of the classifiers say. The accuracy was checked using this final answer and matching it with the actual test labels. **The code is self written.**

The details of the implementation are as follows :

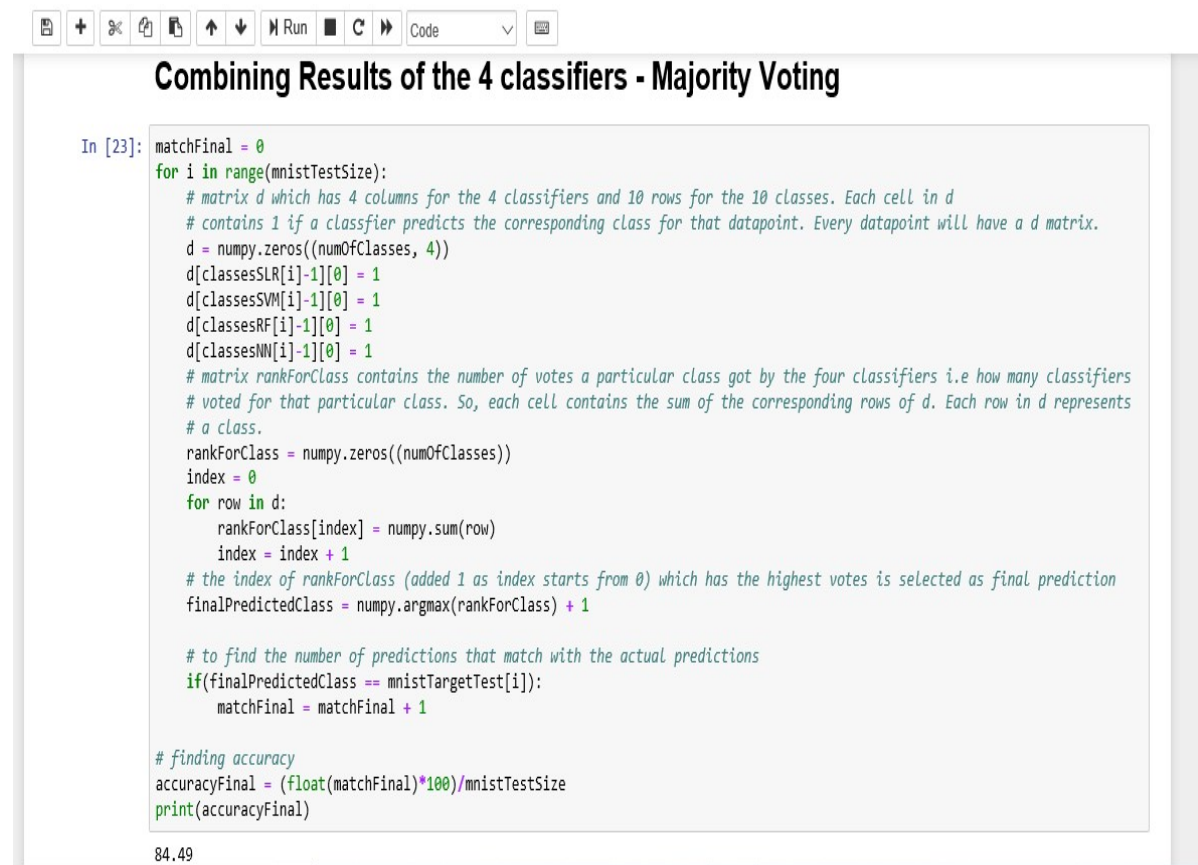
1. A loop was taken which ran through the entire test data set.

2. A matrix called "d" was taken which has 4 columns for the 4 classifiers and 10 rows for the 10 classes. Each cell in "d" contains 1 if a classifier predicts the corresponding class for that data point. Every data point will have a "d" matrix.
3. A matrix called "rankForClass" was defined which contains the number of votes a particular class got by the four classifiers i.e how many classifiers voted for that particular class. So, each cell contains the sum of the corresponding rows of "d" matrix. Each row in "d" represents a class.
4. The index of "rankForClass" matrix which has the highest votes is selected as final prediction as the index represents the class number (one was added to the resultant index as the numpy array index numbering starts from 0). To find this numpy.argmax function was used.
5. This resultant class for the particular data point is the output of the combined 4 classifiers.
6. This is matched with the actual test label to find if it matches or not. The count of matches was maintained to find the final accuracy after the entire test dataset is parsed.

## B. RESULTS AND DISCUSSION

### **Combined Accuracy of the four classifiers : 84.49 %**

The above accuracy was obtained by using the best predictions of all the classifiers.



```

In [23]: matchFinal = 0
        for i in range(mnistTestSize):
            # matrix d which has 4 columns for the 4 classifiers and 10 rows for the 10 classes. Each cell in d
            # contains 1 if a classifier predicts the corresponding class for that datapoint. Every datapoint will have a d matrix.
            d = numpy.zeros((numOfClasses, 4))
            d[classesSLR[i]-1][0] = 1
            d[classesSVM[i]-1][0] = 1
            d[classesRF[i]-1][0] = 1
            d[classesNN[i]-1][0] = 1
            # matrix rankForClass contains the number of votes a particular class got by the four classifiers i.e how many classifiers
            # voted for that particular class. So, each cell contains the sum of the corresponding rows of d. Each row in d represents
            # a class.
            rankForClass = numpy.zeros((numOfClasses))
            index = 0
            for row in d:
                rankForClass[index] = numpy.sum(row)
                index = index + 1
            # the index of rankForClass (added 1 as index starts from 0) which has the highest votes is selected as final prediction
            finalPredictedClass = numpy.argmax(rankForClass) + 1

            # to find the number of predictions that match with the actual predictions
            if(finalPredictedClass == mnistTargetTest[i]):
                matchFinal = matchFinal + 1

        # finding accuracy
        accuracyFinal = (float(matchFinal)*100)/mnistTestSize
        print(accuracyFinal)

```

84.49

## VIII. ANSWERS TO QUESTIONS ASKED IN PROJECT 3 DESCRIPTION

1. We test the MNIST trained models on two different test sets: the test set from MNIST and a test set from the USPS data set. Do your results support the “No Free Lunch” theorem?

The “No Free Lunch” theorem states that there is no model that works for every problem i.e. for every possible dataset taken from different arenas. Here, the model was trained on the MNIST data set and tested on MNIST as well as USPS data set. All the classifiers gave better accuracy of prediction for MNIST test data but performed poorly for USPS test data. As they were trained on “MNIST type” data set although new images they could predict well but USPS images are taken in totally different manner and hence our model performs poorly. **The observation supports the “No Free Lunch” theorem stated above.**

2. Observe the confusion matrix of each classifier and describe the relative strengths/weaknesses of each classifier. Which classifier has the overall best performance?

a. Random Forest Confusion Matrix

[ 970	0	1	0	0	1	4	1	2	1]
[	0	1124	2	3	0	1	3	0	1]
[	6	1	998	5	4	0	4	8	6]
[	0	0	9	971	0	9	0	9	3]
[	4	0	0	0	950	0	5	0	4]
[	3	1	1	15	3	856	6	1	5]
[	6	3	0	0	5	4	936	0	4]
[	1	4	20	3	1	1	0	985	2]
[	6	0	5	6	5	7	3	5	927]
[	10	5	2	12	13	5	0	4	3]
[	10	5	2	12	13	5	0	4	3]

b. Softmax Logistic Regression Confusion Matrix

[ 958	0	2	2	1	4	9	1	3	0]
[	0	1103	2	4	1	2	4	1	18]
[	11	7	897	15	15	1	15	18	44]
[	5	1	20	902	1	31	4	15	21]
[	1	4	6	1	909	0	10	1	8]
[	11	4	4	42	11	750	16	10	36]
[	14	3	5	2	13	12	904	1	4]
[	3	19	26	5	10	0	0	927	3]
[	9	9	9	28	8	22	14	14	848]
[	11	8	5	11	42	12	0	23	6]
[	11	8	5	11	42	12	0	23	6]

c. Neural Networks Confusion Matrix

969	0	0	1	0	3	3	3	1	0
0	1117	4	4	0	0	1	3	6	0
9	1	970	8	6	3	6	12	14	3
1	0	15	945	0	15	0	13	18	3
0	1	2	0	930	1	9	4	3	32
7	0	2	37	2	804	8	4	15	13
8	1	5	0	8	12	917	2	5	0
1	7	20	5	7	0	0	969	1	18
6	3	4	10	6	16	7	5	907	10
6	8	1	6	24	5	1	9	12	937

d. SVM Confusion Matrix

974	0	1	0	0	1	1	1	2	0
0	1129	2	1	0	1	0	1	1	0
4	0	1014	0	1	0	1	7	4	1
0	0	2	997	0	2	0	4	3	2
0	0	2	0	966	0	4	0	1	9
2	0	0	8	1	872	4	0	3	2
4	2	0	0	2	3	946	0	1	0
0	5	8	1	1	0	0	1005	1	7
3	0	2	4	4	2	1	2	953	3
4	2	1	6	8	2	0	6	1	979

From the above confusion we can draw some analysis about how the four classifiers performed:

- The Random Forest Classifier has maximum values in the diagonal which signifies that it predicted most of digits in the images right. What other information that we get is that, the most wrongly predicted number is 7. The number 7 was predicted to be digit 2, 20 times. Even the digit 9 was predicted wrongly to be 1, 10 times. The digit 4 was predicted to be 9, 19 times. We can see that 2 and 7, 1 and 9 when hand written look pretty similar to even human eye many times. Low false negatives and false positive values.
- Softmax Logistic Regression classifier has maximum values in the diagonal which signifies that it predicted most of digits in the images right. But we can see that as compared to all other confusion matrices of other classifiers it has the least numbers in the diagonals. It has highest false negative values with 2 classified as 8, 44 times and when compared with other classifiers. Even, the false positives are high in comparison to other classifiers.
- Neural Networks classifier has maximum values in the diagonal which signifies that it predicted most of digits in the images right. It has higher false negatives than false positives. The false negatives are higher than Random Forest Classifier.
- SVM classifier has maximum values in the diagonal which signifies that it predicted most of digits in the images right. It has extremely low false positives and negatives. When compared with all the other classifiers it has the least false positives and negatives.

**The best classifier individually is the SVM with 98.35% accuracy** followed by Random Forest with 97.16% accuracy followed by Neural Networks with an accuracy of 94.64% and lastly Softmax Logistic Regression with an accuracy of 91.14%.



3. **Combine the results of the individual classifiers using a classifier combination method such as majority voting. Is the overall combined performance better than that of any individual classifier?**

Answered in Sector VII (B)

## IX. RESULTS AND CONCLUSIONS

Classifier	Best Accuracy for MNIST data set	Best Accuracy for USPS data set
SVM	98.35 %	43.19 %
Random Forest	97.16 %	38.63 %
Neural Networks	94.64 %	21.89 %
Softmax Logistic Regression	91.14 %	9.88 %
4-Combined classifier	84.49 %	-

We can see from the above results that SVM performed the best among all the four classifiers. Even, Random Forest classifier did pretty well and is just 1.19% less accurate than SVM. The Softmax Logistic Regression performed the worst at 91.14 % accuracy.

We can also see that the Combined classifier developed by combining the best predictions of all these 4 classifiers performs worse than all the individual classifiers. A reason might be that majority voting might not be an optimal method of combining the classifiers.

Even for USPS data SVM performs the best and Softmax Logistic Regression performs the worst as per the results in the table above.

\*\*\*\*\*