MNIST Digits - Machine Learning Project - Final

April 30, 2020

1 Introduction

`Machine Learning Project`

Comparison of Naive Bayes,Support Vector Machine, Random Forest, and Convolution Neural Network classifiers

`Introduction`

In machine learning, classification is a supervised learning approach in which the computer learns from one or more given inputs, makes observations, and then predictions on one or more outcomes. To be more specific, it draws a conclusion on the input data or training data, and predicts categories or class labels on the output data or given testing data. Some examples of classification problems are spam detection, image segmentation, sentiment analysis, digit recognition, and others. The are various classification algorithms. Some of them are decision tree, logistic regression, multi layer perceptron, support vector machine, random forest, naive bayes, neural networks,and k nearest neighbors. There is not a algorithm better than other. It all depends on the data we are trying to build our model on.

The end goal of this project is to, at a high level, build and evaluate different classifiers, such as Naive Bayes, Support Vector Machines, Random Forest, and Convolution Neural Network classifiers in terms of classifier performance and generalization. In order to do so, we will be using an image dataset called MNIST hand-written digits. This is a Multi Class classification problem in which we are given 28 x 28 pixel images of a hand-written digits ranging from 0 to 9, and we have to classify the images appropriately. In order to that, we will get familiar with the dataset, use hyper parameter tunning to optimize model performance, evaluate the performance of each classifier using a classification report, accuracy classification score, and confusion matrix to evaluate the accuracy of the classification, evaluate predictions on test data set,and compare classifiers performance. The library requirements for this analysis will be NumP, Matplotlib, Scikit-learn, and TensorFlow. We hypothesize that due to excellent reputation of the model for image classification, Convolution Neural Network will provide the lower error rate, which means that it be the optimal classifier to use.

2 Data Manipulation and Visualization

`Data Description`

The MNIST hand-written digits, which stands for Modified National Institute of Standards and Technology, is a popular dataset used for pattern recognition. It has a total of 7000 images. The training set has 6,000 examples and the training set has 1,000. The images are gray-scale, 28 x 28 pixel and centered. Each pixel is related to a number, where 0 represents the dark pixels and 255 represents the white pixels. There are many ways in which you can obtain this dataset. For this project, we decided to obtain it from Keras, which is an open-source Neural Network API written in Python and that can be easily use with TensorFlow. MNIST dataset was originally created by re-mixing samples from a dataset called NIST, which stands for National Institute of Standards and Technology. The hand-written digits were taken from Census Bureau employees and High School students. #This dataset was later normalized to fit 28 by 28 grayscale pixels.

Data Inspection and Visualization

In order to inspect and visualize the dataset, we are going to be using 1 module and 3 libraries. Tensorflow.keras.datasets is a module that allow us to import Keras embeded datasets,in this case MNIST. Keras is a popular deep learning API that builds into TensorFlow and offers functionalities such as tensor.data. Numpy, in order to create a numpy array to represent the bins or number values. Seaborn and matplotlib, in order to plot the value count of the training set.

Importing the MNIST dataset

The first step is to load the dataset and this can be done with a single line of code. We unpack it on the same line of code that will return a tuple of numpy array in the form of (x_train, y_train), (x_test, y_test). The dataset is already shuffled and clean, so there is no need to inspect it for missing values. The next step is to reshape the pixel values to 28 x 28. Last, we print the training and testing set partition.

Visualizing the dataset

In order to visualize the dataset, we will plot a histogram of the value counts for each number in the training set using seaborn and matplotlib.The purpose of doing this is to checked if the dataset is biased towards a particular value. We will first set the number of bins using np.arange(11). Even though, we will get 10 bins regarless if we specified it or not. Doing so, will make sure the numbers for each value are perfectly align in the center of the bins. Then, we will use plt.hist to create a histogram of the value counts. Last, we will set the x and y label and sticks.

Visualizing the digits

In order to visualize the digits, we will create a for loop to obtain the first 9 digits of the training set. Then, we will define the subplot size and title. Last, we will use plt.imshow to plot the first 9 images of the training set,and pyplot.get_cmap to get the images in grayscale.

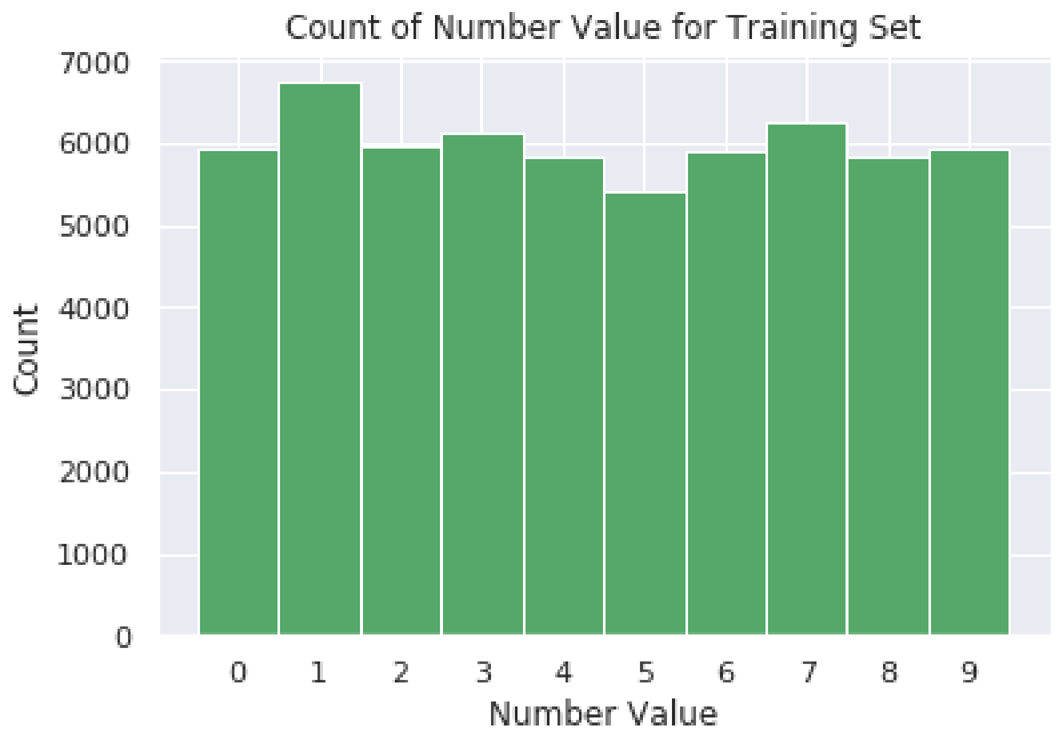
[77]:
```python
#Importing Relevant Libraries and Modules for Data Exploration and Visualization
from tensorflow.keras.datasets import mnist
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib import pyplot
import random
#Expliting Dataset Into Training and Testing
```

```
(x_train,y_train),(x_te, y_te) = mnist.load_data()
#Reshaping Dataset Pixel Value
x_train = x_train.reshape((-1, 28*28))
x_te = x_te.reshape((-1, 28*28))
#Mnist.shape
print('Training Set Partition: X=',(x_train.shape))
print('Testing Set Partition: X=',(x_te.shape))
```

```
Training Set Partition: X= (60000, 784)
Testing Set Partition: X= (10000, 784)
```

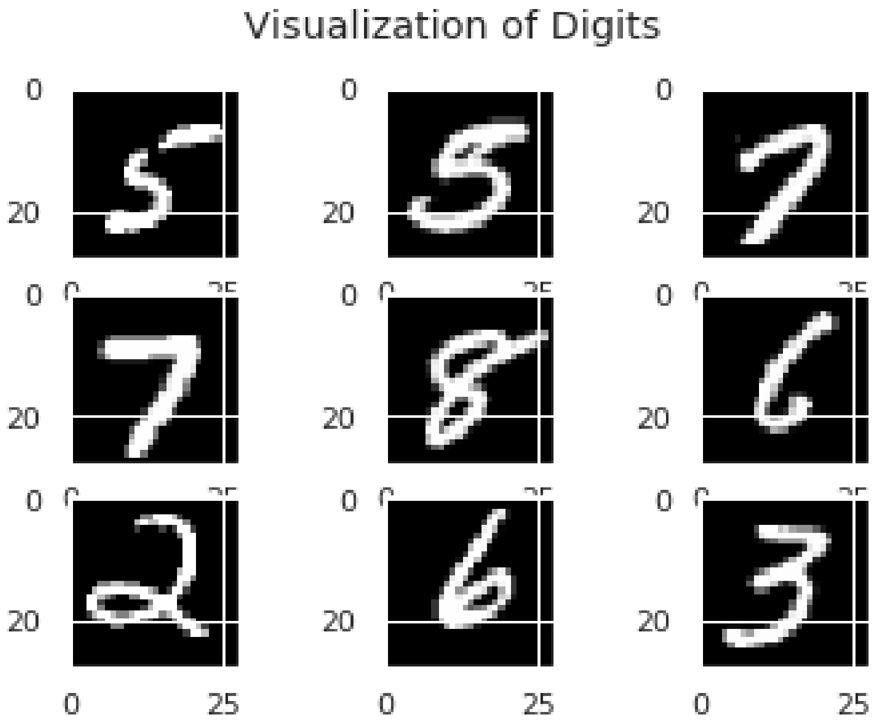
Interpretation of results: As we can observed, this dataset contains 70,000 rows or images and 784 columns. Each of these columns contain a number ranging from 0 255. The training set contains 60,000 images and the testing set 10,000. We didn't have to specified the number of values on the partition, since this slip is already embedded when we import the dataset through tensor.keras.

[78]:
```
#Plotting Training Dataset
sns.set()
bins = np.arange(11) - 0.5
_ = plt.hist(y_train,density=False, facecolor='g', bins = bins)
plt.xlabel('Number Value')
plt.ylabel ('Count')
plt.xticks(range(10))
plt.xlim([-1, 10])
plt.title ('Count of Number Value for Training Set')
plt.show()
```



Interpretation of results: Now, lets inspect if the number or value distribution is even for the training set. The most frequent value is the number 1 followed by 7, so we can say that the training set is biased towards the digit 1 compare to number 5. Unfortunately, this number is also over represented on the testing set.

[79]:
```python
# plotting first few images
for i in range(9):
# Defining subplot
    plt.subplot(330 + 1 + i)
    plt.suptitle('Visualization of Digits')
# Plotting raw pixel data
    plt.imshow(x_tr[i].reshape((28,28)), cmap=pyplot.get_cmap('gray'))
# Showing the figure
plt.show()
```



Interpretation of results: Here we can visualize 9 images from the training set. The images are in gray scale in the shape of 28 x 28. Data Preparation

Splitting the data into training, validation and testing set

In previous steps, we created our training set consisting of 60,000 digits and the testing set consisting of 10,000 digits. Now we further slip the training data into train and validation. We will use 20

- The training data will be used for training the model. - The validation data will be used to tune the parameters and evaluate the model. - The testing data will be used to test the model.

Normalizing Data

As mentioned previously, the pixel values range from 0 to 255. We need to normalize the data dimension so they are in the same scale. There are many way to do this. We can used the tensor.keras.utils or we can just divide the training, validation,and testing set by 255.

[82]:
```
#Importing Relevant Libraries and Modules for Data Partition
from sklearn.model_selection import train_test_split
#Expliting Training Set Into Validation to be Used for CNN Classifier
x_tr, x_val, y_tr, y_val = train_test_split(x_train,y_train, test_size=0.2,␣
 ↪random_state=1,stratify= y_train)
#Summarizing Dataset and Plotting Values
print('Training Set Partition: X=%s, y=%s' % (x_tr.shape, y_tr.shape))
print('Testing Set Partition: X=%s, y=%s' % (x_te.shape, y_te.shape))
print('Validation Set Partition: X=%s, y=%s' % (x_val.shape, y_val.shape))
```

```
Training Set Partition: X=(48000, 784), y=(48000,)
Testing Set Partition: X=(10000, 784), y=(10000,)
Validation Set Partition: X=(12000, 784), y=(12000,)
```

[83]:
```
#Normalizing X Training/Validating/Testing Set to range 0-1 Since Pixel Values␣
 ↪Range from 0 - 255
x_tr = x_tr/255.0
x_te = x_te/255.0
x_val = x_val /255.0
```

3 Feature Engineering

`Dimensionality Reduction with PCA`

Dimensionality reduction is the process of reducing the amount of features in a dataset without losing relevant information for the learning task. The reasons why more features is bad is because, they might be redundant, can be hard to interpret and visualize, can cause fitting problems, reduce noise on data, and can be computationally expensive. Some of the methods for dimensionality reduction are LDA and PCA. LDA is supervised and PCA is unsupervised. PCA considers the entire structure of the data while LDA tends to maximize separation using class information. For the purpose of this analysis, we will be using PCA even though we already know the class labels.

To implement method, we will be:

1) Importing Relevant Libraries:

   We will be importing different functions from the scikit-learn library. This machine learning open-source supports various classification, regression, and clustering algorithms as well as data processing functions. To be able to perform a PCA, we need to import the functions

PCA and StandardScaler from the modules sklearn.decomposition and sklearn.preprocessing. The function PCA will be used as the main method to obtain the number of components, and StandardScaler will be used to make sure the data is internally consistent.

2) Fitting, Transforming the Data and Obtaining number of Components:

   After importing relevant libraries and assigning a variable to StandardScaler, we will fit and transform the standard scaler on the training set, and then apply PCA function. We will not be printing out the number of eigenvectors, since they are not useful for this analysis.

3) Visualizing the PCA Spectrum:

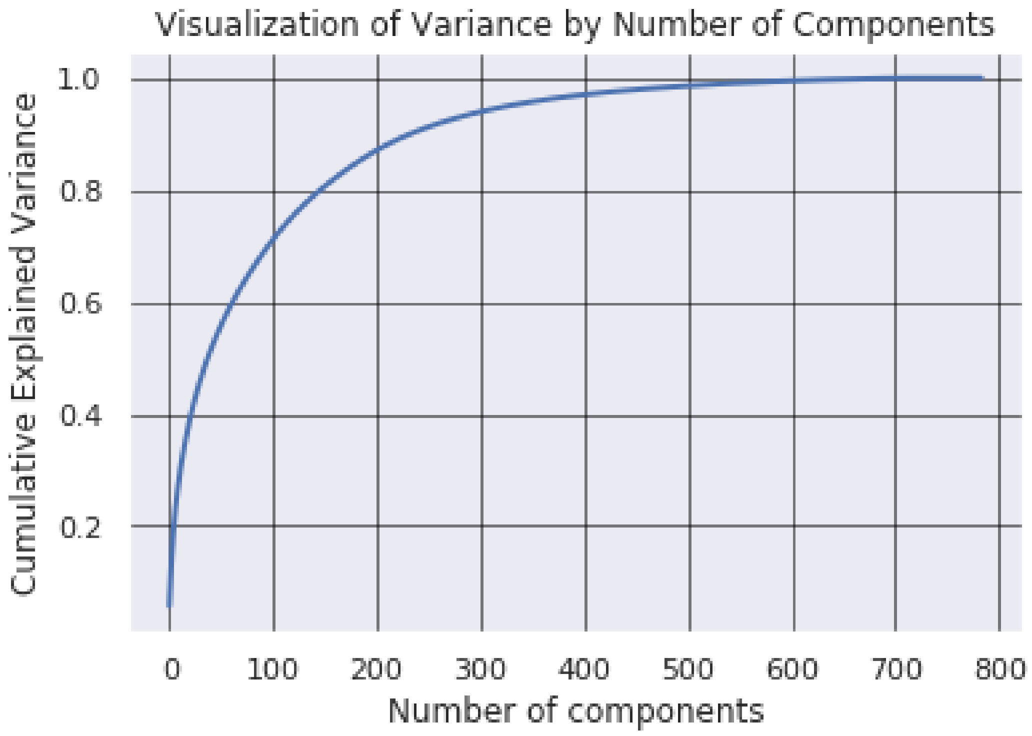
   In order to visualize the cumulative variance by number of principal components, we will use the .cumsum and .explained_variance_ratio_ functions. This will help us understand the number of components and how much of the data these components will explain.

4) Fitting and Transforming on N Components

   After visualizing and deciding the number of components, we will fit and transform the data on the specified number of components.

[84]:
```
#Dimensionality Reduction with PCA
#Importing Relevant Libraries
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
#Fitting and tranforming Data
X_train = sc.fit_transform(x_tr)
pca = PCA().fit(X_train)
#print(pca.components_)
```

[90]:
```
# Plotting the PCA spectrum
fig, ax = plt.subplots()
plt.plot(np.cumsum(pca.explained_variance_ratio_),linewidth=2)
plt.axis('tight')
plt.xlabel('Number of components')
plt.ylabel('Cumulative Explained Variance')
plt.title("Visualization of Variance by Number of Components")
#Setting axis below data
ax.set_axisbelow(True)
# Customizing the grid
ax.grid(linestyle='-', linewidth='0.5', color='black')
plt.show()
```



Interpretation of results: As previously discussed, the plot above help us to understand the proportion of variance explain by number of components. Since we have a total of 784 features, there is a total of 784 components. Ideally, we want to select the number of components that will explain 90% to 95% of the variance, but this depends of the data, the reason why we are using PCA, and where does the explainability significantly decrease. The reason why we don't want to choose a low number is that, even though, we want to reduce the number of features, we don't want to reduce it to a point in which our classifiers can't accurately classify the images and predict the class labels. Based on the plot, 200 components explain about 90% of the variance, 250 components explain about 95% of the variance, and after 450 components, there is not a significant increase.

```
[ ]: #Fitting and Transforming on 250 Components
     #pca = PCA(n_components=250).fit(X_train)
     #X_train = pca.transform(X_train)
     #print("---Our new data using first 250 components---")
     #print(X_train)
```

Interpretation of results: Please note that the reason why we have commented out this portion of the code is that we will not be implementing PCA for this analysis. Due to the high number of samples, it is time consuming and doesn't improve the results significantly.

`Final dataset`

The MNIST dataset is ready for learning. We have reshaped it to a 2 dimensional array of 28 x 28 pixels, normalized it so each pixel values ranges from 0 - 1, and partitioned into training, validation and testing set. The training set contains 48000 images, the testing set partition contains 10000 images, and the validation set 12000.

4 Analysis

`Running Models`

`Support Vector Machine Classifier`

Support Vector Machine is a powerful algorithm considered an extension of perceptron, which goal is to maximize the distance between the decision boundary or hyperplane. It was originally created to work on binary data, but nowadays it is capable of performing multi-class classification. This supervised learning method can be used for classification and regression. The reason for the popularity of this algorithm is that it is capable of solving linear and non linear problems thanks to its kernels.

The advantages of Support Vector Machine are:

- Effective in high dimensional spaces.
- Still effective in cases where the number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines are:

- If the number of features is much greater than the number of samples, avoid over-fitting in choosing Kernel functions and regularization term is crucial.
  - SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see Scores and probabilities, below).

To implement this classifier, we will be:

1) Importing Relevant Libraries:

   We will import different functions from the scikit-learn library. Some of the functions that we will need are choice to create a sample dataset to train the parameters, svm and metrics to set our classifier,
   gridsearchcv to find the optimal hyperparameters of the model which will provide the most accurate
   predictions, classification_report and confusion_matrix to measure the quality of the predictions and describe the performance of the model.

2) Creating a Sample Dataset:

   To reduce the amount time for the model to find the right parameters, we will create a sample dataset from
   the training data consisting of 3,000 images.

3) Setting Grid Search:

As mentioned before, this function allow us to find the optimal parameters for the model. These parameters are passed as arguments to the constructor of the estimator classes. For this classifier, the search will consist of the estimator svm, the parameters C, gamma and kernel,the scoring function accuracy, and the cross-validation scheme. C represents the cost of misclassification. A large C gives you low bias and high variance, and a small C gives you higher bias and lower variance. Gamma is the cut-off parameter for the Gaussian sphere. A large gamma value gives you tighter and bumpier decision boundary which means higher bias and low variance. A small gamma value gives you looser decision boundary which means low bias and high variance. Kernels are algorithms for pattern analysis. Since SVM was originally created for linear separable data, kernels allow us to use this classifier on nonlinear data. We will set 3 different kernels linear, polynomial and Gaussian.

4) Fitting the Sampled Dataset on the Parameters:

   To predict the best parameters, we will fit the gridsearchcv previously set to the sampled dataset using the .fit function.

5) Outputting Results:

   We will use the functions best_score_ and best_params_ to obtain the cross-validated score and parameters of the best estimator and results.

6) Creating a Function to Plot Learning Curve and Plotting Learning Curve:

   We will be creating a function and plotting the learning curve in order to confirm that the best parameters obtained are not overfitting on the validation set. We want to make sure that our validation score is less than the training score. To do that, we will implement a user defined function provided by sklearn (url: https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html).

7) Fitting Best Estimator on Training Set and Predicting on Testing Set:

   After outputting the best parameters, we will fit those parameters on the entire training set, and then make predictions from the model on the testing data. To interpret the success of the model predictions, we will interpret the accuracy scores, classification report and confusion matrix.

8) Plotting Confusion Matrix:

   For a better visualization of the performance of the model, we will print graph the confusion matrix.

(The information about the advantages and disadvantages of SVM was copied word to word from https://scikit-learn.org/stable/modules/svm.html)

[68]:
```
##### Support Vector Machine #####
#Importing Relevant Libraries and Modules
from numpy.random import choice
from sklearn import svm, metrics
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, confusion_matrix
import pandas as pd
#Creating Sample Dataset to Obtain the Right Paramaters
```

```
sample_ind = choice(range(len(x_tr)), 3000)
x_tr_sampled = x_tr[sample_ind]
y_tr_sampled = y_tr[sample_ind]

#Setting Grib Search
c_range = [0.1, 1, 3,5,10]
gamma_range = [10**-3,10**-4, 10**-5, 10**-6]
param_grid = {'C': c_range,
    'gamma': gamma_range,
    'kernel': ["poly", "rbf", "linear"]
             }
# Setting up search
svc = svm.SVC(random_state=2)
gs = GridSearchCV(estimator=svc,
    param_grid=param_grid,
    scoring='accuracy',
    cv=5,
    n_jobs=-1,
    verbose=5)
```

[69]:
```
# Fitting Sampled Model Based on Params
gs = gs.fit(x_tr_sampled, y_tr_sampled)
```

```
Fitting 5 folds for each of 60 candidates, totalling 300 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  48 tasks      | elapsed:   38.2s
[Parallel(n_jobs=-1)]: Done 138 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-1)]: Done 264 tasks      | elapsed:  2.7min
[Parallel(n_jobs=-1)]: Done 300 out of 300 | elapsed:  3.1min finished
```

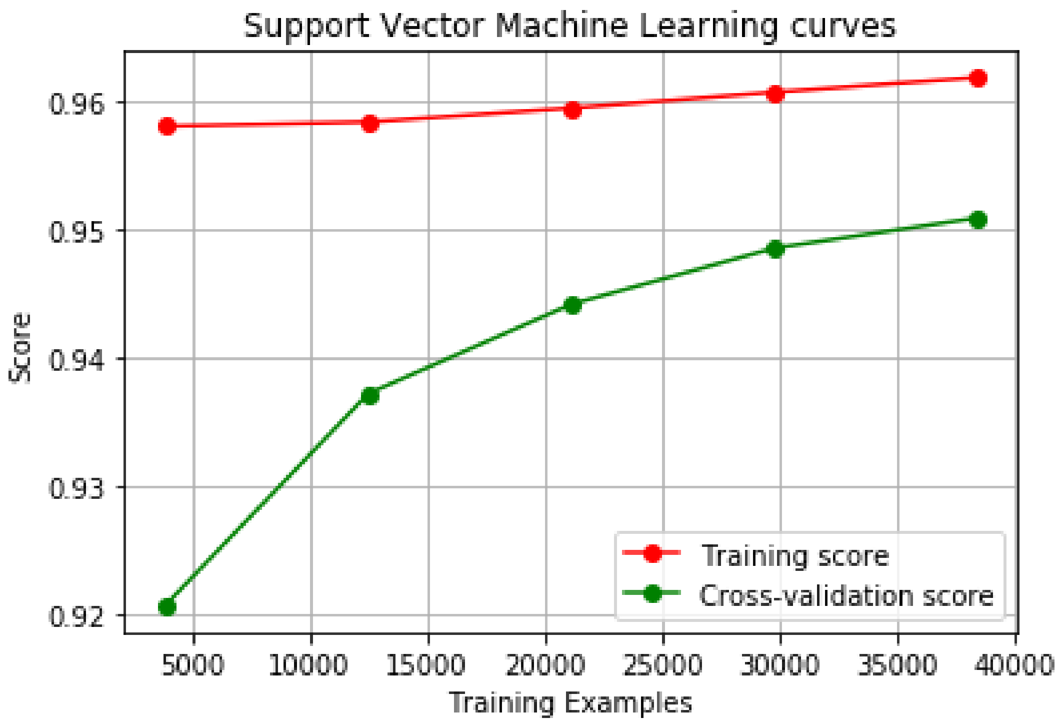
[70]:
```
# Outputting results
print(gs.best_score_)
print(gs.best_params_)
```

```
0.9119999999999999
{'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
```

Interpretation of results: A 91% accuracy was obtained with 3,000 images. I expect this score to increase when we fit the model in the entire dataset. The best parameters were C: 10, gamma of 0.001, and rbf kernel.

[71]:
```
#Creating function to plot learning curve
from sklearn.model_selection import learning_curve
def plot_learning_curve(estimator, title, X, y, ylim=None, cv=None,
                        n_jobs=-1, train_sizes=np.linspace(.1, 1.0, 5)):
#Generating training and validation dataset learning curve."""
    plt.figure()
    plt.title(title)
    if ylim is not None:
        plt.ylim(*ylim)
    plt.xlabel("Training Examples")
    plt.ylabel("Score")
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes)
    train_scores_mean = np.mean(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    plt.grid()
    plt.plot(train_sizes, train_scores_mean, 'o-', color="r",label="Training␣
 ↪score")
    plt.plot(train_sizes, test_scores_mean, 'o-',␣
 ↪color="g",label="Cross-validation score")
    plt.legend(loc="best")
    return
```

[72]:
```
#Plotting Learning Curve
plot_learning_curve(gs.best_estimator_,"Support Vector Machine Learning␣
 ↪curves",x_tr,y_tr,cv=5)
```



Interpretation of results: As previously mentioned, the purpose of this graph was to make sure that the parameters obtained are not overfitting or underfitting on the training set. As we can observe, the training line is about the validation line by about 1%. This give us a good idea about the model being able to generalize on the data instead of memorize it. Since the score seem to be pretty good, we will proceed with fitting the model on the testing set.

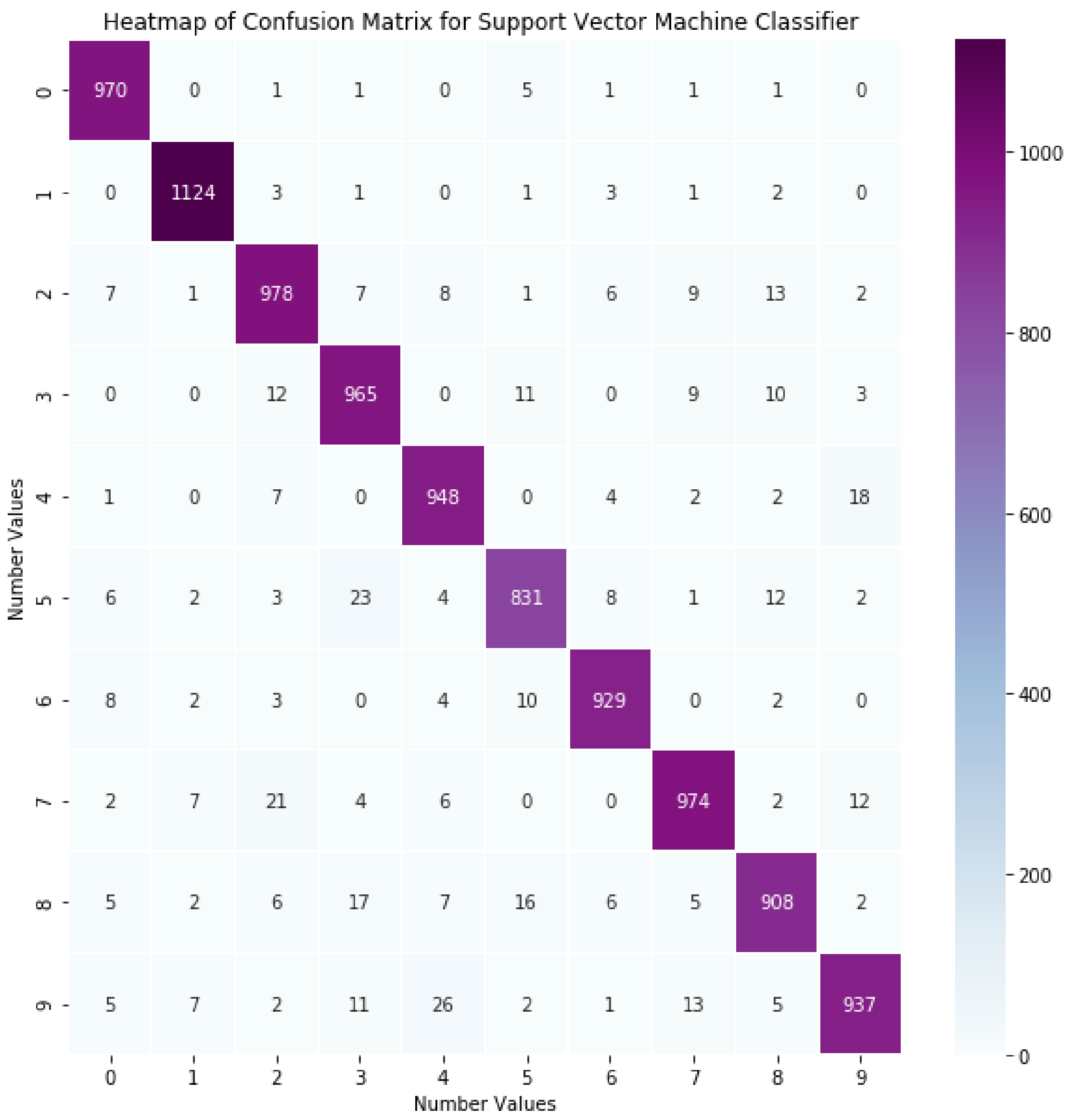
[73]:
```
### Fitting Best Estimator on Training Set and Predicting on Testing Set
svc = svm.SVC(C= 10 ,gamma= 0.001 ,kernel="rbf")
clf = svc.fit(x_tr, y_tr)
print('Accuracy on trainig set: %.2f%% ' % (clf.score(x_tr, y_tr)*100))
svc_pred = clf.predict(x_te)
print(metrics.classification_report(y_te,svc_pred))
#print(confusion_matrix(y_te,svc_pred))
print('Accuracy on testing Set: %.2f%% ' % (svc.score(x_te, y_te)*100))
```

```
Accuracy on trainig set: 96.33%
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       980
           1       0.98      0.99      0.99      1135
           2       0.94      0.95      0.95      1032
           3       0.94      0.96      0.95      1010
           4       0.95      0.97      0.96       982
           5       0.95      0.93      0.94       892
           6       0.97      0.97      0.97       958
           7       0.96      0.95      0.95      1028
           8       0.95      0.93      0.94       974
           9       0.96      0.93      0.94      1009

    accuracy                           0.96     10000
   macro avg       0.96      0.96      0.96     10000
weighted avg       0.96      0.96      0.96     10000

Accuracy on testing Set: 95.64%
```

Interpretation of results: The accuracy score increased from 92% to 96% after fitting the model on the testing set. This means that we were able to predict the right digits 96% of the times or with an accuracy of 96% overall. The model had a good generalization performance. Now we will interpret the recall and precision scores. The recall score means the proportion of correct classifications from cases that are actually positive. For instance, of the 980 zeros on the testing set, 970 were classified correctly or as zeros. The precision score means the proportion of correct classifications from cases that are predicted as positive. For instance, of the 1,04 digits that were classified as zero, only 970 actually were zeros. For the number 1, the model was able to predict this digit with a accuracy of 99%, but the proportion of correct classification was 98% which means that we classified other digits as 1 when they were not 1.For the number 2, we were able to predict this digit with a accuracy of 95%, but the precision score was 94%.For the number 3, we were able to predict this digit with a accuracy of 96%, but the precision score was 94%.For the number 4, we were able to predict this digit with a accuracy of 97%, but the precision score was 95%.For the number 5, we were able to predict this digit with a accuracy of 93%, but the the precision score was 95%.For the number 6, we were able to predict this digit with a accuracy of 97%, but the precision score was 97%. For the number 7, we were able to predict this digit with a accuracy of 95%, but the precision score was

96%. The model classified 41 digits as 7, when they were not 7.For the number 8, we were able to predict this digit with a accuracy of 93%, but the precision score was 95%.For the number 9, we were able to predict this digit with a accuracy of 93%, but the precision score was 96%.

[74]:
```
#Plotting Confusion Matrix
conf_mat_df_svc = pd.DataFrame(confusion_matrix(y_te,svc_pred),␣
 ↪columns=range(10),
                          index=range(10))
#HeatMap
fig, ax = plt.subplots(figsize=(10, 10))
sns.heatmap(conf_mat_df_svc, linewidth = 0.2, annot = True, cmap = 'BuPu',␣
 ↪fmt=".0f")
plt.title('Heatmap of Confusion Matrix for Support Vector Machine Classifier ')
plt.xlabel('Number Values')
plt.ylabel('Number Values')
plt.show()
######### END OF SOLUTION #########
```



	0	1	2	3	4	5	6	7	8	9
0	970	0	1	1	0	5	1	1	1	0
1	0	1124	3	1	0	1	3	1	2	0
2	7	1	978	7	8	1	6	9	13	2
3	0	0	12	965	0	11	0	9	10	3
4	1	0	7	0	948	0	4	2	2	18
5	6	2	3	23	4	831	8	1	12	2
6	8	2	3	0	4	10	929	0	2	0
7	2	7	21	4	6	0	0	974	2	12
8	5	2	6	17	7	16	6	5	908	2
9	5	7	2	11	26	2	1	13	5	937

Interpretation of results: The confusion matrix gives us a better idea of which digits were erroneously classified by others. For instance, 7 digits were classified as zero when they actually were number two. We will the discuss the digits that were highly misclassified for other digits. The model classified 21 two's as seven, 23 five's and 17 eight's as three, 26 nine's as four, 11 three's and 16 eight's as 5, 13 nine's as seven, 13 two's,9 three's and 12 five's as 8, and 18 four's and 12 seven's as nine.

Random Forest Classifier

Random Forest is a combination of multiple decision trees. In other words, it is an ensemble tree-based learning algorithm. An ensemble algorithm combines more than one algorithms for classifying objects then decides the class label (if we use the classifier)through majority voting. This supervised learning method can be used for classification and regression. The random forest classifier creates a set of decision trees from randomly selected subset of training set.

```
The advantages of Random Forest Classifier are:

    - No need for feature scaling
    - Very accurate learning algorithm
    - Efficient on large datasets
    - Has a better generalization performance than an individual decision tree
    - Less sensitive to outliers in the dataset
    - Don't require extensive parameter tuning
    - Can handle a large number of input variables
    - It is good at maintaining accuracy when handling a large number of missing data

The disadvantages of Random Forest Classifier are:
    - Computationally expensive
    - Difficult to visualize
    - Hard to visualize and understand how it reached its conclusion
    - It has been know to overfit for some datasets
```

To implement this classifier, we will be:

1) Importing Relevant Libraries:

   As we did with support vector machine classifier, will import different functions from the scikit-learn
   library. Some of the functions that we will need are RandomForestClassifier to set our classifier, GridSearchCV to find the optimal hyperparameters of the model, and accuracy_score to evaluate the quality of the model.

2) Setting Grid Search:

   As mentioned before, this function allow us to find the optimal parameters for the model. For this classifier, the search will consist of the estimator RandomForestClassifier, the parameters n_estimators, max_depth, min_samples_split, and min_samples_leaf,the scoring function accuracy, and the cross-validation scheme. N estimators represents the number of trees we want to build before taking the average prediction. Max depth specifies the depth of each tree or the number of nodes. Minimum samples split represents the minimum number of samples required to split an internal node. Minimum samples leaf represents the minimum number of samples required to be at a leaf node. Another relevant parameter for random forest is the criterion. The criterion measures the quality of the slip. For this model, we will use the classifier default criterion which is gini. The reason is that I tried both of the possible criterion, gini and entropy, and gini provided the best results. Including this option as part of the parameters, will be very time consuming. In addition, we will set cross validation equal 5. The reason it similar to the criterion. I tried a cross validation of ten, five, and three, and five yielded slightly better results.

3) Fitting the sampled dataset on the parameters:

   To predict the best parameters, we will fit the gridsearchcv previously set to the sampled dataset using the .fit function.

4) Outputting Results:

   We will use the functions best_score_ and best_params_ to obtain the cross-validated score and parameters of the best estimator and results.

5) Plotting Learning Curve:

   We will be plotting the learning curve by using an user defined function (already implement above). The purpose of doing this is to make sure the model is not overfitting or underfitting.

6) Fitting Best Estimator on Training Set and Predicting on Testing Set

   After outputting the best parameters, we will fit those parameters on the entire training set, and then make predictions from the model on the testing data. To interpret the success of the model predictions, we will interpret the accuracy scores, classification report and confusion matrix.

7) Plotting confusion matrix:

   For a better visualization of the performance of the model, we will print graph the confusion matrix.

[57]:
```
###### Random Forest Classifier ######
#Importing Relevant Libraries and Modules
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score
#Setting Grib Search
n_estimators = [100, 300, 500, 800, 1200]
max_depth = [5, 8, 15, 25, 30]
min_samples_split = [2, 5, 10, 15, 100]
min_samples_leaf = [1, 2, 5, 10]

param_grid = {'n_estimators': n_estimators,
              'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
 }

# Setting up search
rfc = RandomForestClassifier(random_state = 4)
rfgs = GridSearchCV(estimator=rfc,
    param_grid=param_grid,
    scoring='accuracy',
    cv=5,
    n_jobs=-1,
    verbose=5)
```

[58]:
```
# Fitting Sampled Model Based on Params
rfgs = rfgs.fit(x_tr_sampled, y_tr_sampled)
```

```
Fitting 5 folds for each of 500 candidates, totalling 2500 fits
```

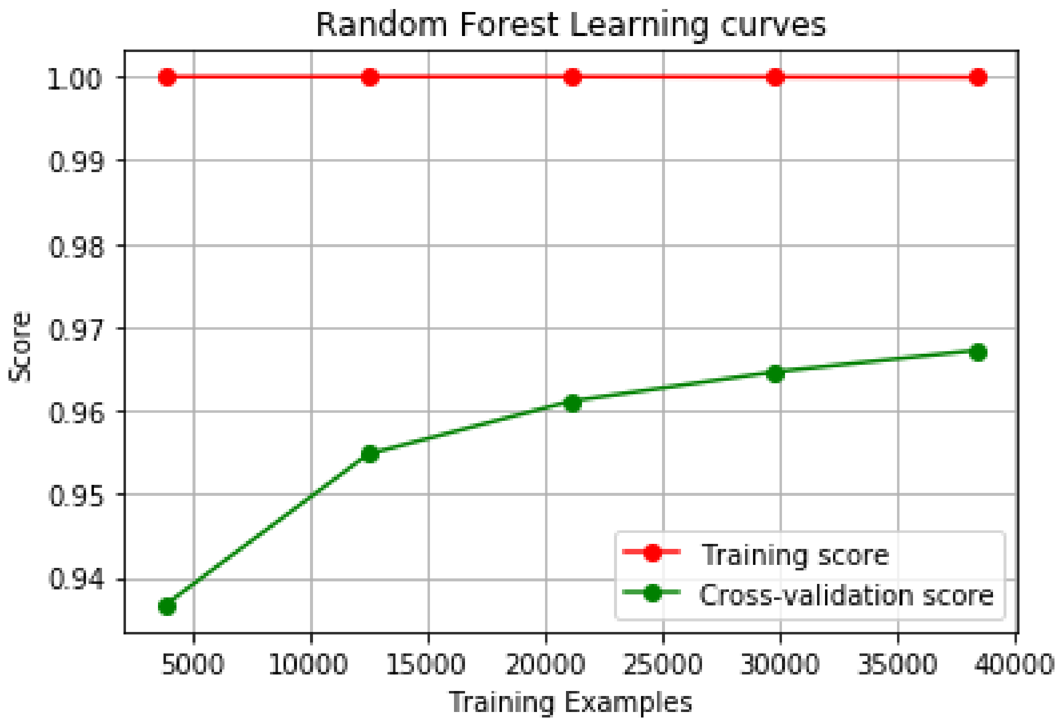
```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  48 tasks      | elapsed:   16.2s
[Parallel(n_jobs=-1)]: Done 138 tasks      | elapsed:   44.1s
[Parallel(n_jobs=-1)]: Done 264 tasks      | elapsed:  1.4min
[Parallel(n_jobs=-1)]: Done 426 tasks      | elapsed:  2.3min
[Parallel(n_jobs=-1)]: Done 624 tasks      | elapsed:  3.6min
[Parallel(n_jobs=-1)]: Done 858 tasks      | elapsed:  5.2min
[Parallel(n_jobs=-1)]: Done 1128 tasks      | elapsed:  7.2min
[Parallel(n_jobs=-1)]: Done 1434 tasks      | elapsed:  9.6min
[Parallel(n_jobs=-1)]: Done 1776 tasks      | elapsed: 12.3min
[Parallel(n_jobs=-1)]: Done 2154 tasks      | elapsed: 15.1min
[Parallel(n_jobs=-1)]: Done 2500 out of 2500 | elapsed: 17.7min finished
```

[59]:
```
# Outputting results
print(rfgs.best_score_)
print(rfgs.best_params_)
```

```
0.9313333333333332
{'max_depth': 25, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators':
800}
```

Interpretation of results: A 93.1% accuracy was obtained with 3,000 images. I expect this score to increase when we fit the model in the entire dataset. The best parameters were a maximum depth of 25, minimum number of samples leaf of 1, minimum samples split of 2, and 1200 estimators.

[61]:
```
#Plotting Learning Rate
plot_learning_curve(rfgs.best_estimator_,"Random Forest Learning␣
↪curves",x_tr,y_tr,cv=5)
```



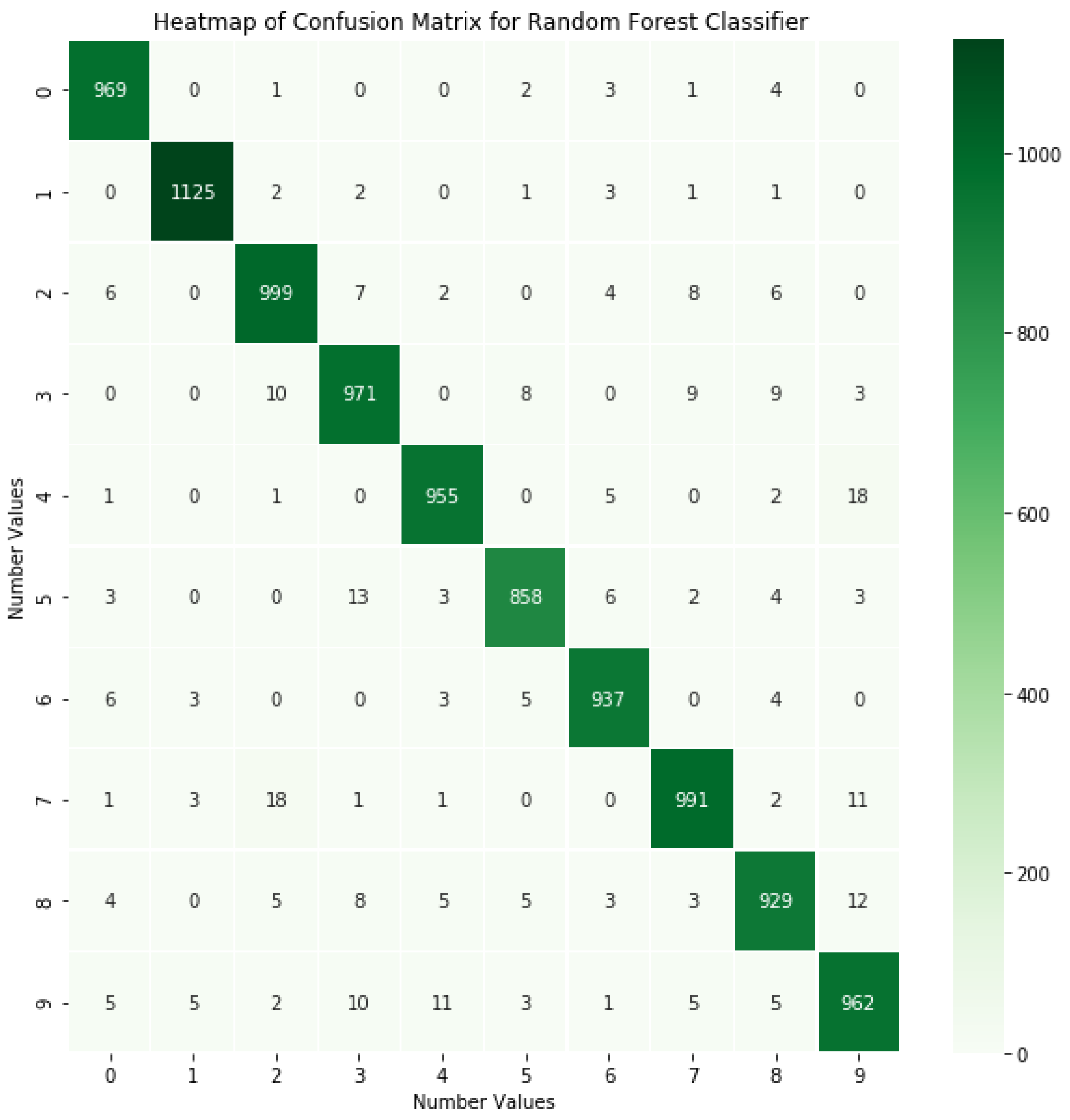
Interpretation of results: As previously mentioned, the purpose of this graph was to make sure that the parameters obtained are not overfitting or underfitting on the training set. Using cross validation usually will prevent the model from overfitting, but we wanted to visually demonstrate this. Even though the training score is almost 100%, our validation score is pretty high, so we will keep the same parameters and expect the test score to be similar to the validation score.

[62]:
```
### Fitting Best Estimator on Training Set and Predicting on Testing Set
rfc = RandomForestClassifier(n_estimators=800,max_depth=25, min_samples_leaf=␣
 ↪1,min_samples_split = 2)
model = rfc.fit(x_tr,y_tr)
print('Accuracy on the Training Set: %.2f%% ' % (model.score(x_tr,y_tr)*100))
rfc_pred = model.predict(x_te)
#print(confusion_matrix(y_te,rfc_pred))
print(metrics.classification_report(y_te,rfc_pred))
print('Accuracy on the Testing Set: %.2f%% ' % (rfc.score(x_te, y_te)*100))
```

```
Accuracy on the Training Set: 99.99%
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       980
           1       0.99      0.99      0.99      1135
           2       0.96      0.97      0.97      1032
           3       0.96      0.96      0.96      1010
           4       0.97      0.97      0.97       982
           5       0.97      0.96      0.97       892
           6       0.97      0.98      0.98       958
           7       0.97      0.96      0.97      1028
           8       0.96      0.95      0.96       974
           9       0.95      0.95      0.95      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000

Accuracy on the Testing Set: 96.96%
```

Interpretation of results: The accuracy score increased from 93.1% to 96.96% after fitting the model on the testing set. This means that we were able to predict the digits 96.96% of the times or with an accuracy of 96.96% overall. As we can see, the model had a good generalization performance. A little bit higher compared to the support vector machine classifier. Now we will interpret the recall and precision scores. Please refer to the svm classifier for more information about how to interpret the precision and recall score. For the number 0, the model was able to predict this digit with a accuracy of 99%, but the proportion of correct classification was 97% which means that we classified other digits as 1 when they were not 0. For the number 1, we were able to predict this digit with a accuracy of 99%, but the precision score was 99% which means that we classified other digits as 1 when they were not 1. For the number 2, we were able to predict this digit with a accuracy of 97%, but the precision score was 96%. For the number 3, we were able to predict this digit with a accuracy and precision score of 96%. For the number 4, we were able to predict this digit with a accuracy and precision score of 97%. For the number 5, we were able to predict this digit with a accuracy of 96%, but the the precision score was 97%. For the number 6, we were able to predict this digit with a accuracy of 98% and a precision score 97%. For the number 7, we were able to predict this digit with a accuracy of 96%, but the precision score was 97%. For the number 8, we were able to predict this digit with a accuracy of 95%, but the precision score was 96%. For the number 9, we were able to predict this digit with a accuracy score of 95% and precision score of 95%.

[75]:
```python
#Plotting Confusion Matrix
conf_mat_df_rfc = pd.DataFrame(confusion_matrix(y_te,rfc_pred),␣
 ↪columns=range(10),
                          index=range(10))
#HeatMap
fig, ax = plt.subplots(figsize=(10, 10))
sns.heatmap(conf_mat_df_rfc, linewidth = 0.2, annot = True, cmap = 'Greens',␣
 ↪fmt=".0f")
plt.title('Heatmap of Confusion Matrix for Random Forest Classifier ')
plt.xlabel('Number Values')
plt.ylabel('Number Values')
plt.show()
######### END OF SOLUTION #########
```



	0	1	2	3	4	5	6	7	8	9
0	969	0	1	0	0	2	3	1	4	0
1	0	1125	2	2	0	1	3	1	1	0
2	6	0	999	7	2	0	4	8	6	0
3	0	0	10	971	0	8	0	9	9	3
4	1	0	1	0	955	0	5	0	2	18
5	3	0	0	13	3	858	6	2	4	3
6	6	3	0	0	3	5	937	0	4	0
7	1	3	18	1	1	0	0	991	2	11
8	4	0	5	8	5	5	3	3	929	12
9	5	5	2	10	11	3	1	5	5	962

Interpretation of results: The confusion matrix gives us a better idea of which digits were erroneously classified by others.We will the discuss the digits that were highly misclassified for other digits. The model classified 10 three's and 18 seven's as two, 13 five's and 10 nine's as three, 11 nine's as four, 9 three's as eight, and 18 four's, 11 seven's and 12 eight's as 9.

`Naive Bayes Classifier`

Naive Bayes is a simple and easy to build classifier which is very useful for large datasets. It's based on Bayes' Theorem with an assumption of independence among predictors. In other words, a Naive Bayes classifier, assumes that the features in a class are not related to other features. This algorithm is very famous for text classification and problems with multiple classes.

The advantages of Naive Bayes Classifier are:

- It is easy and fast to predict class on test data set. It also perform well in multi class prediction
- When assumption of independence holds, a Naive Bayes classifier performs better compare to other models like logistic regression and you need less training data.
- It performs better on categorical input variables than numerical variables . For numerical variable, normal distribution is assumed (bell curve, which is a strong assumption).

The disadvantages of Naive Bayes Classifier are:

- It assumes that the predictors are independent
- It is considered a bad estimator, so the results from predict_proba should not be taken seriously
- The categories in a categorical variable must be represented on both the train set and test set. If not, the model will assign a 0 probability.

To implement this classifier, we will be:

1) Importing Relevant Libraries:

   As we did with previous models, will import a function from the scikit-learn library. We will use GaussianNB to set our classifier.

2) Fitting on the training set and predicting on the testing set:

   Note: For this model, we will not using gridsearch. The reason is that there isn't a hyper-parameter to tune. In addition, we will not be using any cross-validation method, because it will not yield better results. I tried before and the accuracy score stayed the same. Though, I have included the code for Stratified KFold for verification purposes.

   In order to train our model, we will use the fit function on the training set, and then use the predict function on the testing set. At the end, we will use the classification report and accuracy scores to evaluate the model.

3) Plotting confusion matrix:

   For a better visualization of the performance of the model, we will print graph the confusion matrix.

(The information about the advantages and disadvantages of the classifier was obtained from https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/)

[31]:
```python
########## Naive Bayes ##########
#Importing Relevant Libraries and Modules
from sklearn.naive_bayes import GaussianNB
random.seed(7)
#Create a Gaussian Classifier
gnb = GaussianNB()
#Train the model using the training set
gnb.fit(x_tr, y_tr)
#Predict the response for test dataset
gnb_pred = gnb.predict(x_te)
#print(confusion_matrix(y_te,gnb_pred))
```

```
print(metrics.classification_report(y_te,gnb_pred))
print('Accuracy: %.2f%% ' % (gnb.score(x_te, y_te)*100))
```

```
              precision    recall  f1-score   support

           0       0.71      0.94      0.81       980
           1       0.82      0.96      0.89      1135
           2       0.91      0.25      0.39      1032
           3       0.70      0.30      0.42      1010
           4       0.87      0.16      0.27       982
           5       0.54      0.05      0.08       892
           6       0.67      0.92      0.78       958
           7       0.88      0.30      0.44      1028
           8       0.30      0.73      0.43       974
           9       0.40      0.93      0.56      1009

    accuracy                           0.56     10000
   macro avg       0.68      0.55      0.51     10000
weighted avg       0.68      0.56      0.51     10000

Accuracy: 56.13%
```

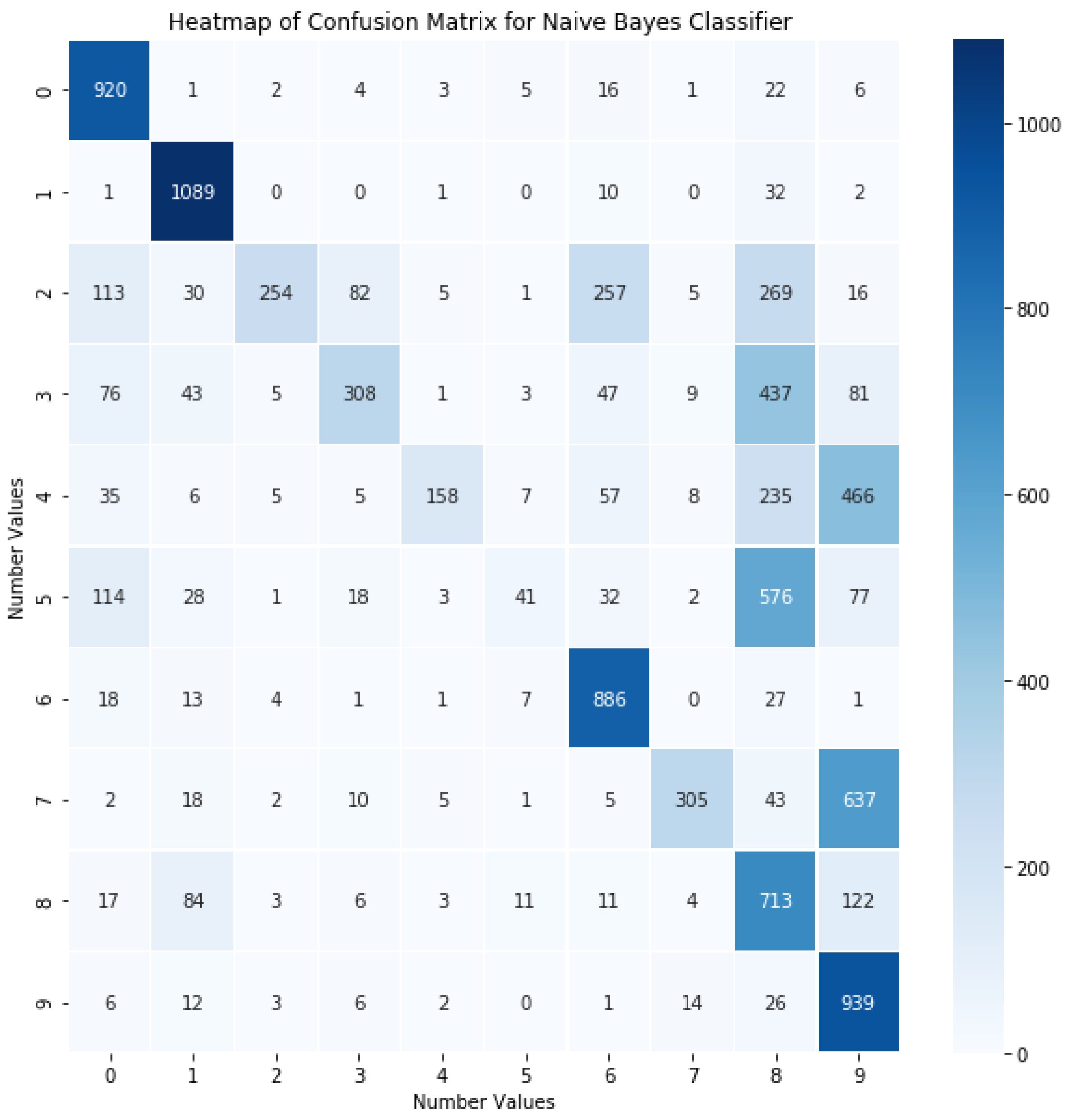
Interpretation of results: The accuracy score is 56.13%. This means that we were able to predict the digits 56.13% of the times or with an accuracy of 56.13%. Overall, the model did very poorly classifying the digits. Now we will interpret the recall and precision scores. For the number 0, we were able to predict this digit with a accuracy of 91%, but the proportion of correct classification was 71% which means that the model classified a large number of digits as 1 when they were not 0. For the number 1, we were able to predict this digit with a accuracy of 96%, but the precision score was 82% which means that we classified other digits as 1 when they were not 1. For the number 2, we were able to predict this digit with a very low accuracy of 25%, but the precision score was 91%.For the number 3, we were able to predict this digit with a very low accuracy of 30%, but the precision score was 70%. For the number 4, we were able to predict this digit with a very low accuracy of 16%, and precision score of 87%. For the number 5, we were able to predict this digit with a accuracy of 5%,the lowest in the entire model, but the the precision score was 54%. For the number 6, we were able to predict this digit with a accuracy of 92% and precision score of 67%. For the number 7, we were able to predict this digit with a accuracy of 30%, but the precision score was 88%. For the number 8, we were able to predict this digit with a accuracy of 73%, but the precision score was 30%.For the number 9, we were able to predict this digit with a accuracy of 93%, and precision score of 40%. One of the reasons for the poor performance of the classifier is that it assumes feature independence.

```
[32]: # Training Model Using Stratified KFold
def eval_classifier_NB(X, y, niter):
    accuracies = []
    kf = StratifiedKFold(n_splits=10,shuffle=False,random_state=None)
    for train_index, test_index in kf.split(X, y):
        clf = GaussianNB()
        model = clf.fit(x_tr[train_index], y_tr[train_index])
        y_pred = clf.predict(x_tr[test_index])
        accuracies.append(accuracy_score(y_tr[test_index], y_pred))
    print('Stratified 10-fold cross validation accuracy is %.3f with %d total␣
 ↪iterations' % (np.mean(accuracies), niter))
eval_classifier_NB(x_tr, y_tr, 10)
```

```
Stratified 10-fold cross validation accuracy is 0.564 with 10 total iterations
```

Interpretation of results: After fitting the model using Stratified KFold with 10 splits, the accuracy score was 56.4%, which means that we were only able to predict correctly 56.4% of the digits. As previously mentioned, there is not a difference compare to the fitting the model without Stratified KFold.

```
[33]: #Plotting Confusion Matrix
conf_mat_df_gnb = pd.DataFrame(confusion_matrix(y_te,gnb_pred),␣
 ↪columns=range(10),
                              index=range(10))
#HeatMap
fig, ax = plt.subplots(figsize=(10, 10))
sns.heatmap(conf_mat_df_gnb, linewidth = 0.2, annot = True, cmap = 'Blues',␣
 ↪fmt=".0f")
plt.title('Heatmap of Confusion Matrix for Naive Bayes Classifier ')
plt.xlabel('Number Values')
plt.ylabel('Number Values')
plt.show()
######### END OF SOLUTION #########
```



Interpretation of results: The confusion matrix gives us a better idea of which digits were erroneously classified by others. Since the accuracy score was very low, we will the discuss the digits that were highly misclassified for other digits. The model confused at a very high rate the numbers two, three and five as 0, the numbers three and eight as 1, the number two as 3, the numbers two, three,four and five as 6, the numbers two, three, four, five and seven as 8, and the numbers three, four, seven and eight as 9.

`Convolutional Neural Network Classifier`

The convolutional neural network (CNN) classifier is a powerful algorithm for image classification. It belongs to the class of deep learning neural networks and was inspired by how the visual cortex of the human brain works when recognizing objects. CNN is mainly composed by 3 different layers: convolutional layer, pooling layer, and a set of fully connected layers. The way in which CNN works is that the very first layer, the convolutional layer, extract the features from the images in the dataset. Convolution is filtering the images with a smaller pixel filter to reduce the image size without losing the relationship between the pixels. The next layer is the pooling layer. The purpose of this layer is decrease the spacial size. The way it works is by selecting a region and then taking the maximum value in that region, and that becomes the new value for the entire region. The last layer is the fully connected layer. The objective of this layer is to take the results from the convolution and pooling layer and use it to classify the images. In order to implement a fully connected layer, we have to flatten the output from the final convolutional or pooling layer, which means that we have to unroll all its values into a vector since they are in 3 dimensional matrix. The basic structure of a CNN is: Convolution -> Pooling -> Convolution -> Pooling -> Fully Connected Layer -> Output.

To implement this classifier, we will be:

1) Importing relevant libraries:

   For this model, we will be using tensorflow.keras, which is a TensorFlow's high-level API for building and training deep learning models. The functions that will be using are Dense, Activation, Flatten, Conv2D, MaxPooling2D and sequential to build our models, and to_categorical to one hot encoding the target values.

2) Reshaping the train, validation, and testing set:

   In order to use the keras API, we need to reshape our dataset into a 4 dimensional numpy array. To do that, we use the .reshape to reshape the dataset, and .astype to make sure the values are float.

3) One Hot Encoding the target values:

   Since each class is represented by a unique value, we will use one hot encoding to convert each integer to an array of 1 or 0. We can successfully run this classifier without this step. We decided to do it, because it is usually recommended.

4) Building the model

   To build the CNN model, we have to think back to the idea behind the layers. Everything we explained at the beginning, will implement now. First, we will start with Sequential(), because this will be a sequential model, in which an instance of the Sequential class is created and model layers are created and added to it. Then, we will build out first layer using the model.add() syntax. We will first start with a 32 nodes convolutional layer with a 3 X 3 window, then we will pass the input shape and padding. Using padding,can improve the performance by keeping information at the borders. This will mainly affect the pixels in the corner. Then, we will use model.add to activate our activation funtions which in this case will be relu. This is the most common activation function for CNN. Last, we will add the pooling layer. We will use max pooling of a 2 by 2 size since the background of the images is black. The only difference between the first and second layer will be the number of layers. For the second layer, we will use 64 nodes. At the end, we will add the fully connected layer. To do this, we will first flatten the vector since we need a 1 dimensional dataset. Then, we will add a dense layer of 64 nodes. Dense layers are the ones that will perform the classification task. Last, we will add softmax to compute the lost function. Softmax activation allow us to calculate the output based on the probabilities. Each class is assigned a probability and the class with the highest probability is the model's output.

5) Compiling the model:

   After setting out layers, we need to compile the model. Compiling is what will allow us to build the model.This is where you define the type of loss function, optimizer and the metrics evaluated by the model during training and validation. In order to do so, we will set loss to categorical_crossentropy. This calculates the error rate between the predicted value and the original value. We will use Adam as our optimizer. The optimizer is in charge of updating the weights of the neurons via backpropagation. It computes the derivative of the loss function with respect to each weight and subtracts it from the weight. Last, we set the metrics to accuracy to report the accuracy of the model.

6) Fitting the model

   This is where we will be fitting the model on the training and validation set. We will set the batch size equal to 64 and the number of epochs to 10. We probably won't need this many epochs, but for visualization purposes, we have decided use it.

7) Obtaining and visualizing the training and validation loss:

   To visualize the Performance of the model for the training and validation set, we will be getting training and validation loss histories by using the .history function. Then we will use matplotlib to visualize the performance of the model.

8) Evaluating the model on the testing set:

   We will evaluate the performance of model on the testing set by using the .evaluate function, print the test lost and accuracy score, and confusion matrix.

9) Results visualization and digits predictions:

   Here, we will be creating a heatmap to better interpret the confusion matrix and visualizing the digits predictions. In order to create the heatmap, we will first calculate the predictions, the we will turn those predictions into a pandas dataframe, and then create the heatmap.

(Resources used: https://blog.tensorflow.org/2018/04/fashion-mnist-with-tfkeras.html) https://medium.com/@ksusorokina/image-classification-with-convolutional-neural-networks-496815db12a8

[34]:
```
########## Convolution Neural Network ##########
#Importing Relevant Libraries and Modules
random.seed(8)
#pip install tensorflow
from tensorflow.keras.layers import Dense, Activation, Flatten, Conv2D,␣
 ↪MaxPooling2D
from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import to_categorical
#Reshaping data
x_tr = x_tr.reshape((-1, 28, 28, 1)).astype('float32')
x_te = x_te.reshape((-1, 28, 28, 1)).astype('float32')
x_val = x_val.reshape((-1, 28, 28, 1)).astype('float32')
x_tr.shape
```

```
#One Hot Encoding
y_tr = to_categorical(y_tr)
y_te = to_categorical(y_te)
y_val = to_categorical(y_val)
#num_classes = 10 y_test.shape[1]
```

[35]:
```
model = Sequential()
#Layer #1
model.add(Conv2D(32, (3,3), input_shape=(28, 28, 1), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
#Layer #2
model.add(Conv2D(64, (3,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

#Adding Artifial Neural Network
model.add(Flatten()) #Converts a 3D to 1D vector
model.add(Dense(64, activation='relu')) #Sums all of the inputs that having␣
 ↪been multiplied by the rate and adding bias function
model.add(Dense(10)) #10 neurons for the classes
model.add(Activation('softmax'))
#model.add(tensorflow.keras.layers.Softmax())
model.summary()
```

```
WARNING:tensorflow:From /home/cpsoler/.local/lib/python3.7/site-
packages/tensorflow/python/ops/init_ops.py:1251: calling
VarianceScaling.__init__ (from tensorflow.python.ops.init_ops) with dtype is
deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the
constructor
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        320
_________________________________________________________________
activation (Activation)      (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 12, 12, 64)        18496
_________________________________________________________________
activation_1 (Activation)    (None, 12, 12, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 2304)              0
_________________________________________________________________
dense (Dense)                (None, 64)                147520
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
_________________________________________________________________
activation_2 (Activation)    (None, 10)                0
=================================================================
Total params: 166,986
Trainable params: 166,986
Non-trainable params: 0
_________________________________________________________________
```

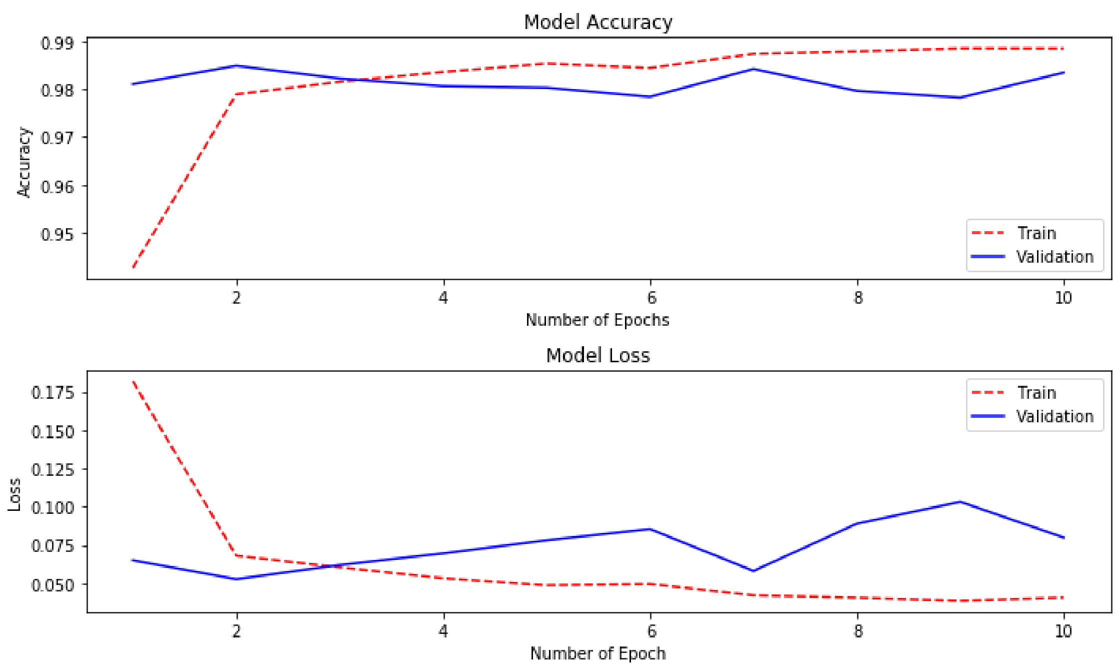
[36]:
```
#Compiling Model
import tensorflow
model.compile(loss=tensorflow.keras.losses.categorical_crossentropy,
              optimizer=tensorflow.keras.optimizers.Adam(lr=0.
 ↪01),metrics=['accuracy'])
```

[37]:
```
#Fitting Model
history_fit = model.fit(x_tr, y_tr, batch_size=64, epochs=10,␣
 ↪validation_data=(x_val, y_val))
```

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/10
48000/48000 [==============================] - 8s 160us/sample - loss: 0.1818 -
acc: 0.9426 - val_loss: 0.0651 - val_acc: 0.9811
Epoch 2/10
48000/48000 [==============================] - 7s 154us/sample - loss: 0.0681 -
acc: 0.9789 - val_loss: 0.0528 - val_acc: 0.9849
Epoch 3/10
48000/48000 [==============================] - 8s 156us/sample - loss: 0.0605 -
acc: 0.9816 - val_loss: 0.0620 - val_acc: 0.9822
Epoch 4/10
48000/48000 [==============================] - 8s 158us/sample - loss: 0.0534 -
acc: 0.9836 - val_loss: 0.0696 - val_acc: 0.9807
Epoch 5/10
48000/48000 [==============================] - 7s 155us/sample - loss: 0.0488 -
acc: 0.9854 - val_loss: 0.0781 - val_acc: 0.9803
Epoch 6/10
48000/48000 [==============================] - 7s 154us/sample - loss: 0.0496 -
acc: 0.9845 - val_loss: 0.0854 - val_acc: 0.9784
Epoch 7/10
48000/48000 [==============================] - 7s 154us/sample - loss: 0.0423 -
acc: 0.9874 - val_loss: 0.0581 - val_acc: 0.9843
Epoch 8/10
48000/48000 [==============================] - 7s 155us/sample - loss: 0.0407 -
```

```
acc: 0.9879 - val_loss: 0.0890 - val_acc: 0.9797
Epoch 9/10
48000/48000 [==============================] - 7s 154us/sample - loss: 0.0388 -
acc: 0.9886 - val_loss: 0.1032 - val_acc: 0.9783
Epoch 10/10
48000/48000 [==============================] - 7s 153us/sample - loss: 0.0408 -
acc: 0.9885 - val_loss: 0.0799 - val_acc: 0.9835
```

[38]:
```python
# Obtaining and visualizing the training and validation loss
# Creating Count of the Number of Epochs
training_loss = history_fit.history['loss']
epoch_count = range(1, len(training_loss) + 1)
# Plotting the metrics
fig, ax = plt.subplots(figsize=(10, 6))
plt.subplot(2,1,1)
plt.plot(epoch_count,history_fit.history['acc'], 'r--')
plt.plot(epoch_count, history_fit.history['val_acc'],'b-')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Number of Epochs')
plt.legend(['Train', 'Validation'], loc='lower right')

plt.subplot(2,1,2)
plt.plot(epoch_count,history_fit.history['loss'],'r--')
plt.plot(epoch_count,history_fit.history['val_loss'],'b-')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Number of Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.tight_layout()
```



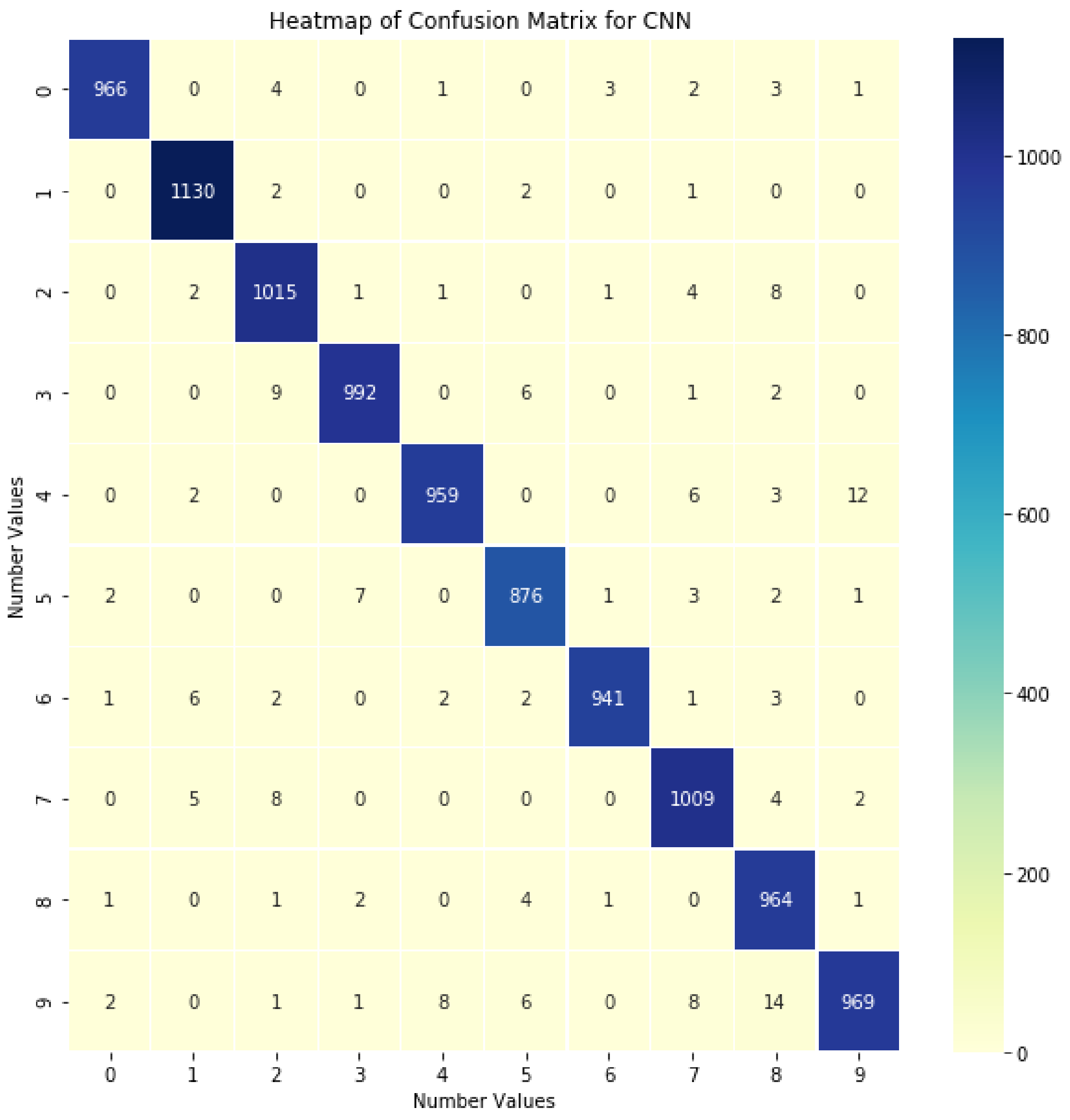
Interpretation of results: After 10 epochs, we obtained a validation accuracy of 98.35%. This shows the ability of the model to generalize on the new data. We can also observed that the highest accuracy was achieved at 8 epochs. In the subsequent epochs, the accuracy doesn't improve. The validation accuracy and training accuracy are pretty close but not touching, this is an indicator that the model is performing well. If the training accuracy keeps improving while your validation accuracy gets worse, the model is overfitting, which means that the model is memorizing the data instead of learning from it. In addition, the validating loss seems to fluctuate between .06 to .08 and back to .06. This seems to be okay but if the model fluctuates a lot,it might also be an indicator that it is memorizing instead of learning.

[39]:
```
#Evaluating Model on Testing Set
eval_score = model.evaluate(x_te, y_te, verbose=0)
print('Test loss: %.2f%% ' % (eval_score[0]*100))
print('Test accuracy: %.2f%%'% (eval_score[1]*100))
```

```
Test accuracy: 98.21%
```

Interpretation of results: After fitting the model on the testing set, the accuracy score was 98.21%. This means that we were able to predict the digits 98.21% of the times or with an accuracy of 98.20% overall.

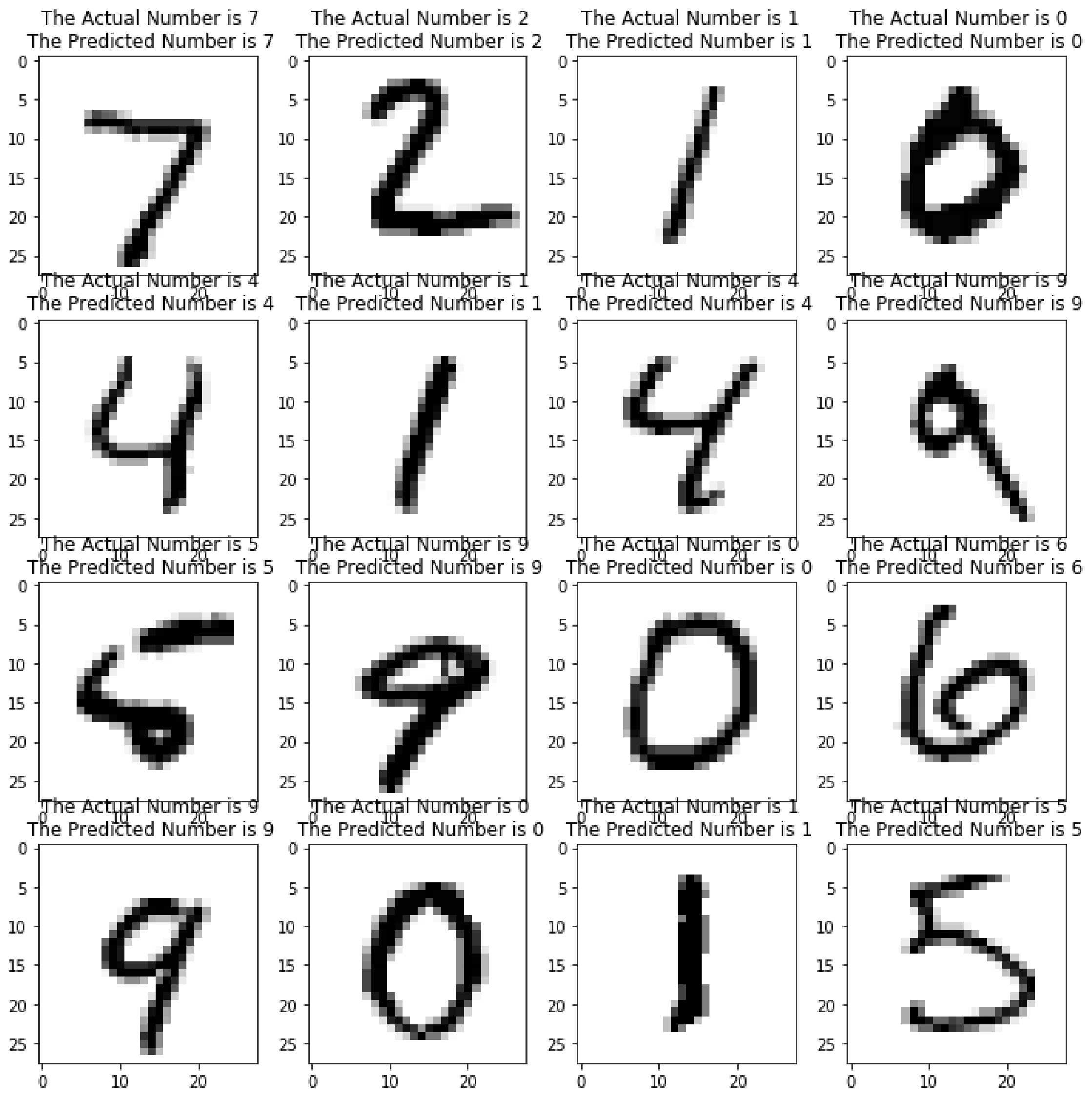
[40]:
```
#Obtaining Confusion Matrix
cnn_pred = model.predict(x_te)
#print(confusion_matrix(y_te.argmax(axis=1),cnn_pred.argmax(axis=1)))
# Creating Pandas DataFrame to Plot Confusion Matrix
conf_mat_df_cnn = pd.DataFrame(confusion_matrix(y_te.argmax(axis=1),cnn_pred.
 ↪argmax(axis=1)),
                                        columns=range(10),index=range(10))
#Plotting Confusion Matrix - HeatMap
fig, ax = plt.subplots(figsize=(10, 10))
sns.heatmap(conf_mat_df_cnn, linewidth = 0.2, annot = True, cmap = 'YlGnBu',␣
 ↪fmt=".0f")
plt.title('Heatmap of Confusion Matrix for CNN ')
plt.xlabel('Number Values')
plt.ylabel('Number Values')
plt.show()
```



	0	1	2	3	4	5	6	7	8	9
0	966	0	4	0	1	0	3	2	3	1
1	0	1130	2	0	0	2	0	1	0	0
2	0	2	1015	1	1	0	1	4	8	0
3	0	0	9	992	0	6	0	1	2	0
4	0	2	0	0	959	0	0	6	3	12
5	2	0	0	7	0	876	1	3	2	1
6	1	6	2	0	2	2	941	1	3	0
7	0	5	8	0	0	0	0	1009	4	2
8	1	0	1	2	0	4	1	0	964	1
9	2	0	1	1	8	6	0	8	14	969

Interpretation of results: The confusion matrix gives us a better idea of which digits were erroneously classified by others. We will the discuss the digits that were highly misclassified for other digits. The model confused at a the number six as 0, the number four as 1, the number seven as 2, the numbers five as 3, the number nine as 4, and the numbers four and seven as 9.

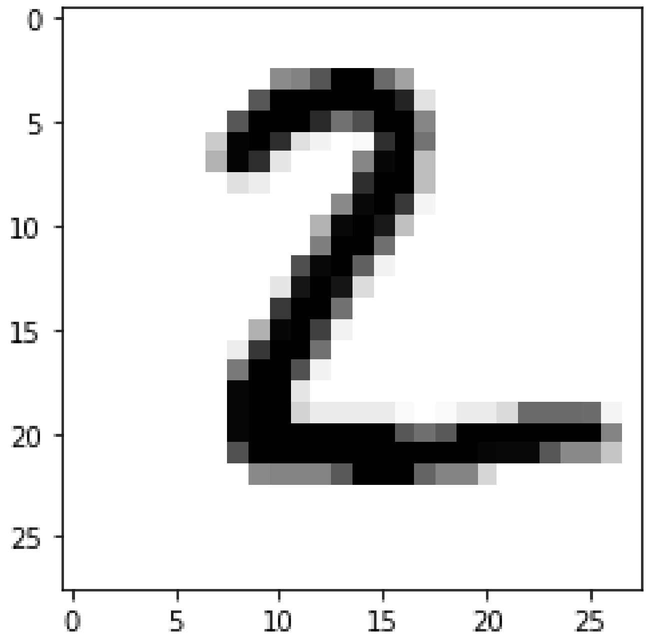
```
[41]: #Visualizing of All Predictions Predictions
      X_test_Numbers = x_te.reshape(x_te.shape[0], 28, 28)

      fig, axis = plt.subplots(4, 4, figsize=(12, 12))
      for i, ax in enumerate(axis.flat):
          ax.imshow(X_test_Numbers[i], cmap='binary')
          ax.set(title = f" The Actual Number is {y_te[i].argmax()}\n The Predicted␣
       ↪Number is {cnn_pred[i].argmax()}");
```



```
[42]: #Printing Single Predictions
      print(np.argmax(np.round(cnn_pred[1])))
```

```
2
```

```
[43]: #Visualizing Single Predictions
      plt.imshow(x_te[1].reshape(28, 28), cmap = plt.cm.binary)
      plt.show()
      ######### END OF SOLUTION #########
```



Interpretation of results: We can observe a combination of multiple and single predictions. This is what the model predicted and what it actually is.

5 Conclusion and Ethics

`Discussion and Future Work`

The end goal of this project was to compare,at a high level, Naive Bayes,Support Vector Machine, Random Forest, and Convolution Neural Network in terms of classifier performance and generalization. We were able to successfully accomplish this goal by trying multiple hyperparameters and using those parameters to build the models. We later analyzed the results for each classifier in terms of its performance. Now, we will summarize the overall performance and generalization of the models. The term generalization applies to the ability of the model to don't over fit or under fit on the new data, and the term performance refers to the ability of the model to accurately predict the classes on the unseen data. One of the best ways to evaluate a classifier performance and generalization is to interpret the accuracy score for the training/validation and testing set, classification report, and confusion matrix. We used MNIST Digits datatset to perform the analysis. This popular dataset has total 70,000 images and 784 features. In order to get our data ready for the models, we normalized and reshape the data pixels, split the dataset into training, validation and testing, and demonstrated how to perform dimensionality reduction. Overall, all the classifiers with the exception of Naive Bayes had a good generalization and performance. CNN gave us the best results with a accuracy score of 98.37% on the validation set and 98.20% on the testing set, followed by Random Forest with a accuracy score of 93% on the sampled validation set and 97% on the testing set. Support Vector Machines resulted on a accuracy score of 92% on the sampled validation set and 96% on the testing set.

As we hypothesize, CNN had the highest accuracy score, and we can say that this is due to the nature of the model. Please note that as mentioned above, we used a sampled dataset to obtain the best parameters and the scores for these parameters are the ones that we are providing as validation/testing accuracy scores. Having said that, this might be the reason why the accuracy score for the testing set is higher compared to the validation set. Even though, 3 out of the 4 classifiers had a good performance by being able to predict on unseen data with a accuracy ranging 96.82% to 98.20%, these classifiers still have run for improvement. For instance, some of the recommendations we have to improve the convolution neural network classifier are to try to increase the number of dense layers and reduce the number of epochs or remove the dense layer and increase the number of convolutional layers. We can find the best combination by creating a for loop with different parameters and then build the model using the sequential function. Please note that this computationally expensive. We can improve the performance on the Random Forest Classifier by reducing the number of parameters this can prevent the model from memorizing on the training data.

Ethical Implications

There are some ethical implications related to the field of image processing which humans have little difficulty with. This implications are related to human labor and privacy. One of the controversies is associated to the reduction of human labor since this task is highly perform by humans, we could expect a number of companies relaying on machines to perform the task of digits and facial recognition. Another implication is the privacy concern linked to the process of facial recognition.