# Report I - SF2957 Statistical Machine Learning

Anna Barysheva 960603-7423 Karin Larsson 980522-4384
Ruxue Zeng 981015-T549

2 December, 2021

## 1 Classification of MNIST

### 1.1 Objectives

In this project, we will use three different methods to design and implement a classifier for handwritten digit recognition : Stochastic subgradients, Gaussian process and Convolutional neural network. The dataset is made up of 1797 images of size $8 \times 8$. Each image is of a low-resolution hand-written digit.

### 1.2 Mathematical Background

In this section, we describe the general set up of our multi-class support vector machine.

#### 1.2.1 $R^{emp}$ convex

We consider a classification problem with $K = 10$ classes $\{0, 1, ..., 9\}$ and $d = 8 \times 8 = 64$. The classifier is represented using matrix

$$\Theta = [\theta_1 \theta_2 \dots \theta_K] \in \mathbb{R}^{d \times K}$$

Given $\Theta$ the predicted class for a data vector image $x \in \mathbb{R}^d$ is the label

$$\hat{y} = \underset{j=1,\dots,K}{\text{argmax}} [\Theta^T x]_j$$

To train the classifier, we minimize the empirical risk using a multi-class hinge-loss

$$L(\Theta, (x, y)) = \max_{j \neq y} \Big[1 + \sum_{i=1}^{d} x_i (\Theta_{ij} - \Theta_{iy})\Big]_+,$$

where $t_+ = max(t, 0)$ is the positive part. More precisely, we aim to minimize

$$R^{emp}(\Theta) = \frac{1}{n} \sum_{k=1}^{n} L(\Theta, (x_k, y_k))$$

We want to give a subgradient G of $R^{emp}(\Theta)$. To do so, firstly, we must to show that $R^{emp}(\Theta)$ is convex.
Let $\lambda \in [0,1]$ and $(\Theta, \Psi) \in \mathbb{R}^{d\times K} \times \mathbb{R}^{d\times K}$, we have:

$$R^{emp}(\lambda\Theta + (1-\lambda)\Psi) = \frac{1}{n}\sum_{k=1}^{n} \max_{j\neq y_k} \Big(1 + \sum_{i=1}^{d} x_{k,i}(\lambda\Theta_{ij} + (1-\lambda)\Psi_{ij} - \lambda\Theta_{iy_k} - (1-\lambda)\Psi_{iy_k})\Big)$$

We rewrite $1 = \lambda + (1-\lambda)$ and separate the sum in two parts, we obtain:

$$R^{emp}(\lambda\Theta+(1-\lambda)\Psi) = \frac{1}{n}\sum_{k=1}^{n} \max_{j\neq y_k} \Big(\lambda + \sum_{i=1}^{d} x_{k,i}(\lambda\Theta_{ij} - \lambda\Theta_{iy_k}) + (1-\lambda) + \sum_{i=1}^{d} x_{k,i}((1-\lambda)\Psi_{ij} - (1-\lambda)\Psi_{iy_k})\Big)$$

Since the maximum of a sum is less or equal to the sum of the maximum, we obtain the inequality :

$$\begin{aligned} R^{emp}(\lambda\Theta + (1-\lambda)\Psi) \leq \frac{1}{n}\Bigg[ & \sum_{k=1}^{n} \max_{j\neq y_k} \Big(\lambda + \sum_{i=1}^{d} x_{k,i}(\lambda\Theta_{ij} - \lambda\Theta_{iy_k})\Big) \\ & + \sum_{k=1}^{n} \max_{j\neq y_k} \Big((1-\lambda) + \sum_{i=1}^{d} x_{k,i}((1-\lambda)\Psi_{ij} - (1-\lambda)\Psi_{iy_k})\Big)\Bigg] \end{aligned}$$

Thus, we have :

$$\boxed{R^{emp}(\lambda\Theta + (1-\lambda)\Psi) \leq \lambda R^{emp}(\Theta) + (1-\lambda)R^{emp}(\Psi)}$$

So $R^{emp}$ is a convex function, now we can calculate its subgradient.

### 1.2.2 Subgradient $G$ of $R^{emp}$

We assume we have only one data set $(x, y)$, thus $L(\Theta, (x, y)) = R^{emp}(\Theta)$. We consider

$$j^* = \max_{j\neq y} \Big[1 + \sum_{i=1}^{d} x_i(\Theta_{ij} - \Theta_{iy})\Big]_+ , \quad L(\Theta, (x,y)) = \Big[1 + \sum_{i=1}^{d} x_i(\Theta_{ij^*} - \Theta_{iy})\Big]_+$$

We have two cases:

- $L(\Theta, (x, y)) = 0$ : which means $1 + \sum_{i=1}^{d} x_i(\Theta_{ij^*} - \Theta_{iy}) \leq 0 \ \forall j$. If the inequality is strict, the expression is differentiable, we have gradient equals to zero matrix. If we have the equality, we can take the simplest subgradient zero matrix.Thus, we have

$$G = 0 \in \mathbb{R}^{d\times k}$$

- $L(\Theta, (x, y)) > 0$ : which means $L(\Theta, (x, y)) = 1 + \sum_{i=1}^{d} x_i(\Theta_{ij^*} - \Theta_{iy})$. We take the derivative of this expression with regard to $j^{th}$ column, we obtain :

$$G = \begin{cases} x & \text{if } j = j^* \\ -x & \text{if } j = y \\ 0 & \text{otherwise} \end{cases}$$

### 1.3 Results

#### 1.3.1 Stochastic Subgradient

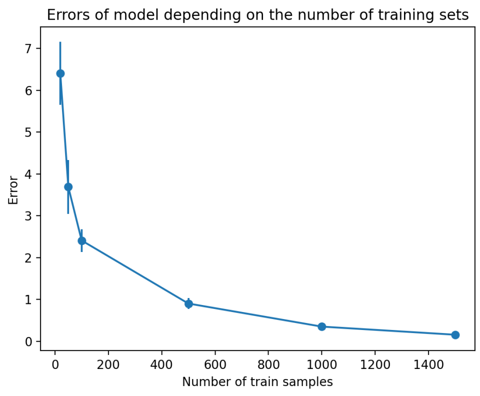
Using the subgradient calculated of $R^{emp}$, a projected stochastic gradient descent classifier was implemented to classify the MNIST dataset.

Initially, we took the stepsize $\epsilon_k$ proportional to $k^{-\alpha}$ for $\alpha \in (0,1]$, and project $\Theta$ to the ball

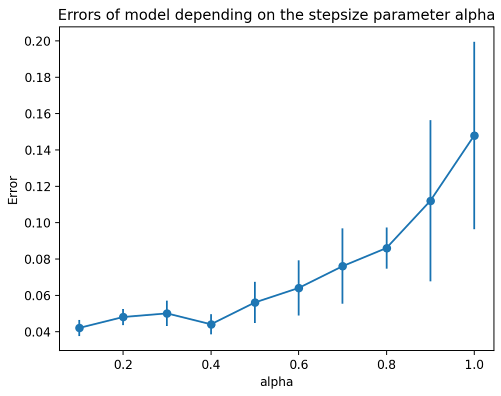
$$B_r = \{\Theta \in \mathbb{R}^{dk} : \sum_{i,j} \Theta_{ij}^2 \leq r^2\}$$

We took $\alpha = 0.5$ and evaluated the performance of the classifier using training sets of size 20, 50, 100, 500, 1000 and 1500, and a test set of size 100. We plot the error and the standard deviation over 30 trials in figure 1a. We observed the classifier gets a better performance when increasing the number of training sets.

Then, with 1697 training sets and 100 test sets, we investigated the performance of the classifier with different choices of $\alpha$. We observe in figure 1b that the performance of the classifier gets worse when increasing the $\alpha$. We remark also that the standard derivation become extremely large for $\alpha \in (0.6, 1]$.



(a) With different size of training set



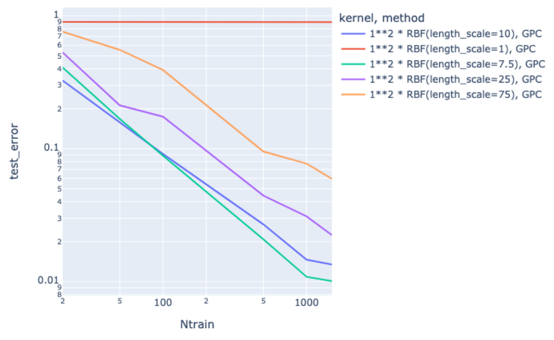
(b) With different stepsize parameter $\alpha$

Figure 1: Error rate and SGD on the test set of size 100

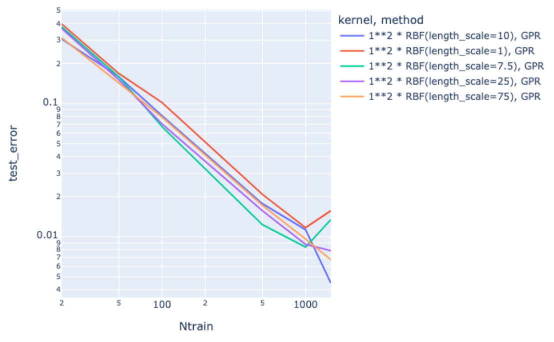
### 1.3.2 Gaussian process

A Gaussian process classifier was implemented to classify the MNIST dataset using two different methods. Firstly the scikit learn method GaussianProcessClassifier (GPC) was used. Secondly the scikit learn GaussianProcessRegressor (GPR) was used with $y$ as a one hot vector. The performance was evaluated by analysing the mean of the relative error of the test set. The mean was computed over 3 tries. This was done for multiple kernels. In order to be able to compare the same kernel on GPC and GPR, GPC (optimize = None) was used.

When using a radial basis function (RBF) as the kernel, the length scale $L$ was of high importance to the error rate. This can be seen for some parameters for the GPC in figure 2a and for the GPR in figure 2b. GPR seems to be less affected by the choice of $L$, and gave lower error rates for most points, see figure 3



(a) Using GPC



(b) Using GPR

Figure 2: Error rate on the test set using RBF with some different parameters

Since RBF gave a reasonably good error rate, the more general Matern kernel was also tested with different values of $\nu$. $\nu$ can be seen as a smoothness parameter where the larger $\nu$, the smoother the resulting function has to be. As $\nu \to \infty$ Matern()$\to$ RBF(), therefore it is reasonable that the error values of Matern() approaches the values of RBF() as $\nu$ increases. The Matern kernel was applied with different $\nu$ and compared to the corresponding RBF function, as can be seen in figure 4.

When the dot product was used as the kernel, there seemed to be very little difference depending on the sigma parameter. Some results can be observed in figure 5. There is however a big difference between the different methods. GPC (optimize = None) gives a lower error rate than GPR for most of the kernels and number of training elements. For GPR the DotProduct() behaves unexpectedly an increases with some sizes of the training

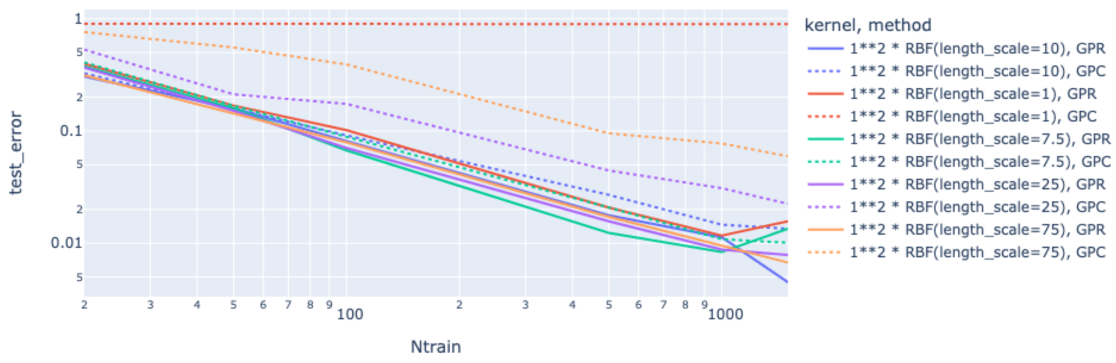
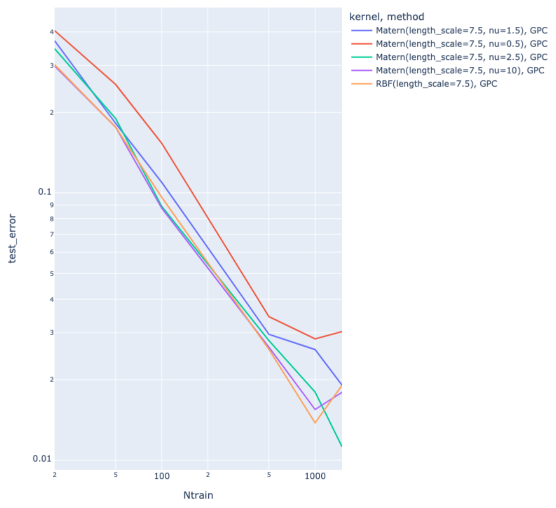
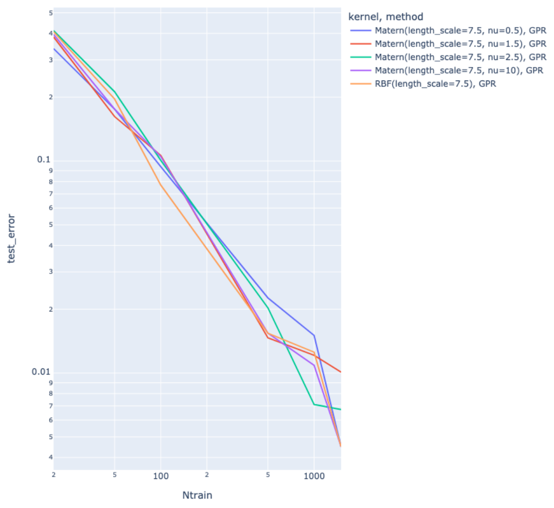


Figure 3: Error rate on RBF



(a) Using GPC

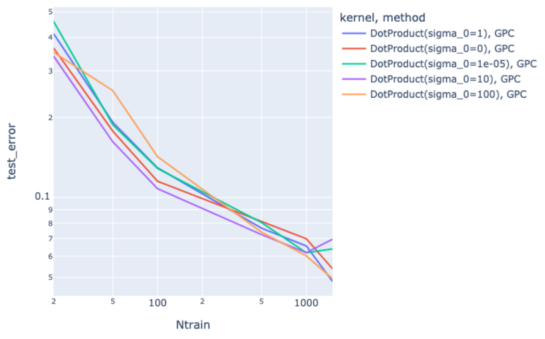


(b) Using GPR

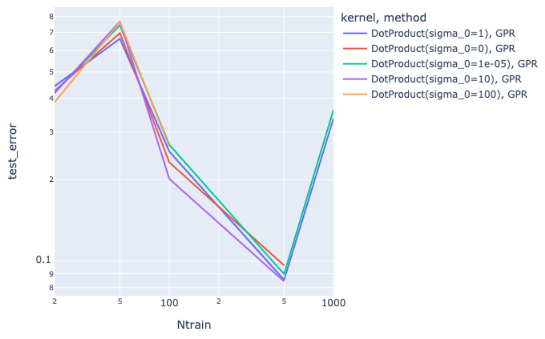
Figure 4: Error rate on the test set using Matern($l = 7.5$) with some different values of $\nu$

set. For large sets of training data and some values $\sigma_0$, the model can not be fitted to the data since the dot product is not returning a positive definite matrix.

Different kernels (with parameters that gave a decent error rate on the test set) are compared in figure 6. Form this it can be observed that RBF ($L = 10$) gives a lower error rate than the dot product and the dot product with an additional noise term. By adding



(a) Using GPC



(b) Using GPR

Figure 5: Error rate on the test set using DotProduct with some different parameters

the noise term the error is slightly lowered, which is to be expected.
When comparing GPC and GPR for these methods (figure 6), it can be observed that the GPC gives a lower error rate on the test set when using the dot product kernel, but GPR gives a lower error rate for the RBF and Matern kernels.

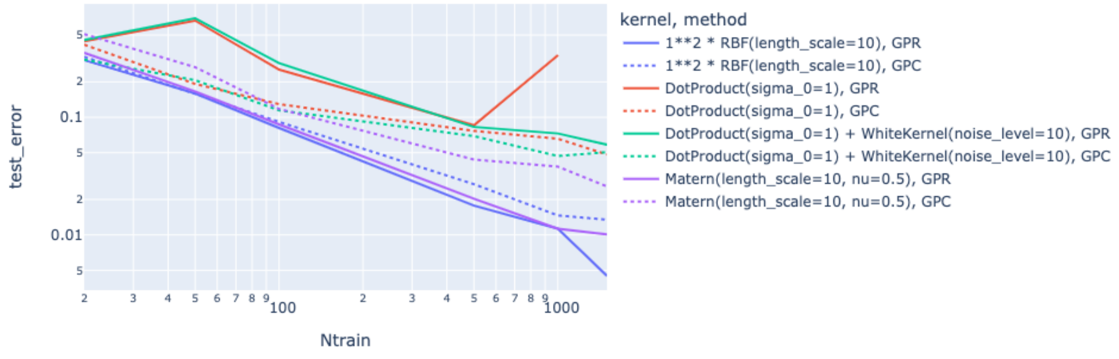


Figure 6: Error rate on test set

As described above the best test error rate was obtained by using the RBF ($L = 10$) kernel closely followed by the Matern kernels ($L = 10$) with the GaussianProcessRegressor being slightly more accurate than the GaussianProcessClassifier. Both methods are pretty computational heavy, however a high precision of classification was achieved. The computation time increased significantly with the size of the training data.

### 1.3.3 Convolutional Neural Network

In the third part of the project, a simple convolutional neural network (CNN) is used to perform the multi-classification of the digits data set using `Tensorflow` and `Keras`. The task is to classify a given image of a handwritten digit into one of 10 classes $(0, 1, ...9)$.

#### Digits classification data set

The `digits` dataset from `sklearn` consists of 1797 small square $8 \times 8$ pixel grayscale images of handwritten single digits. A quarter of data (450 examples) is held as a test data set, the other 75% of data is used to train models.

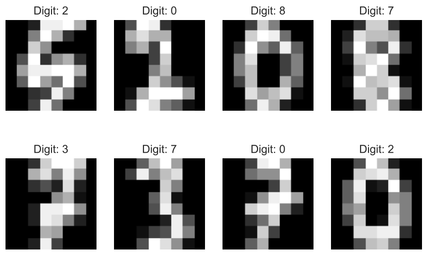


Figure 7: Subset of images from `digits` data set

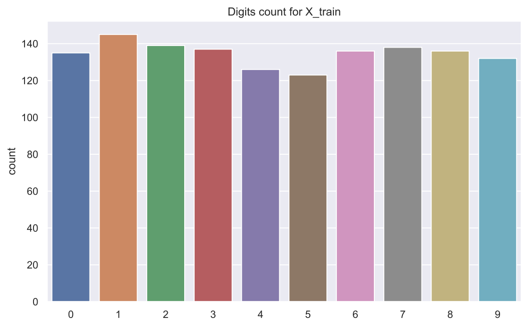


Figure 8: Occurrence of each label in a train data set

Figure 7 shows the first eight images in the training data set. It is important to check if the train data set is balanced, namely the frequency of digits occurrence. From Figure 8, the training set is balanced and thus it can be assumed there is an equal distribution of classes in the training data set.

Since the pixel values for each grayscale image in the data set are unsigned integers in the range from 0 to 255, normalisation of the pixel values is needed. This involves dividing the pixel values by 255 which gives values from 0 to 1.

#### Model validation

In order to estimate the performance of developed models, both 5-fold cross-validation and evaluation on a test data set generated at random are used. k-Fold cross-validation is under consideration since it results in a less unbiased estimate of the models' skill in comparison with a simple train/test split.
A sparse categorical cross-entropy loss and the accuracy as a metric are shown for implemented models.

### Simple NN

To perform mini-batch gradient descent, batch_size is fixed to be 32 for all models. The number of simulation times (epoch) is set to be 100. Moreover, given that the problem is a multi-class classification and for each of 10 output nodes, it is required to assign a probability, a softmax activation function is always used in the output layer.

At first, a one-layer NN is designed as shown below. A hidden layer with 1000 nodes uses ReLU as an activation function. This model has 75,010 trainable parameters and achieves already a good fit with the accuracy of 0.913 on the test set and 0.829 when averaging over 5 folds for cross-validation. Here SGD with the learning rate of 0.01 is used as an optimizer.

```
inputs=keras.Input(shape=(pic_size,1))
fl=Flatten()(inputs)
hidden1=Dense(1000, activation='relu')(fl)
outputs=Dense(10, activation='softmax')(hidden1)
opt=SGD(learning_rate=0.01)
```

The loss and accuracy functions over episodes for a simple one-layer NN are shown on Figure 9. It can be noted that for such a simple neural network, the performance is already outstanding. The loss function rapidly decreases for both train and test data sets and there is no overfitting.

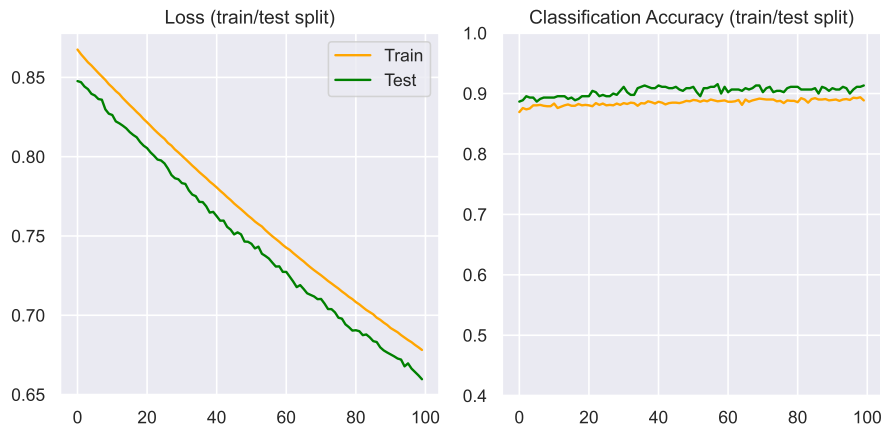


Figure 9: Loss and accuracy for a one-layer CNN, train/test split

To improve it further, a kernel for the activation function is set by the He weight scheme and the momentum method is added to the optimizer.

```
inputs=keras.Input(shape=(pic_size,1))
fl= latten()(inputs)
hidden1=Dense(1000, activation='relu', kernel_initializer='he_uniform')(fl)
outputs=Dense(10, activation='softmax')(hidden1)
opt=SGD(learning_rate=0.01, momentum=0.9)
```

In this case, the average accuracy score on 5-fold cross-validation reaches 0.964 and 0.982 on a test data set which is much better compared to the fisrt model having the same number of parameters. However, the loss function on the test set tends to increase over epoch and diverge from the loss function on a train test. While the training error goes down, the test loss goes up (Figure 10). Thus other models should be studied.

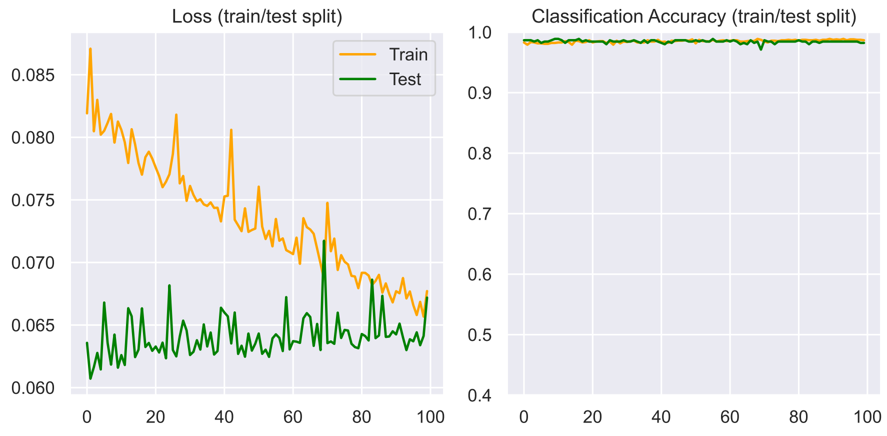


Figure 10: Loss and accuracy for an ungraded one-layer CNN, train/test split

### Multi-layer neural network

Consider a deep neural network with four hidden layers where ReLU is used as an activation function. This NN has 621,160 training parameters and its architecture looks as follows:

```
inputs_deep=keras.Input(shape=(pic_size,1))
fl=Flatten()(inputs_deep)
hidden1=Dense(1000,activation='relu')(fl)
hidden2=Dense(500, activation='relu')(hidden1)
hidden3=Dense(100, activation='relu')(hidden2)
hidden4=Dense(50, activation='relu')(hidden3)
outputs_deep = Dense(10, activation = 'softmax')(hidden4)
opt=SGD(learning_rate=0.01)
```

From Figure 11, it can be concluded this model does not perform correctly and heavily overfits the given data. Since we work with image recognition, it might be a good idea to add convolutional layers and reduce the number of parameters in a model to prevent overfitting.

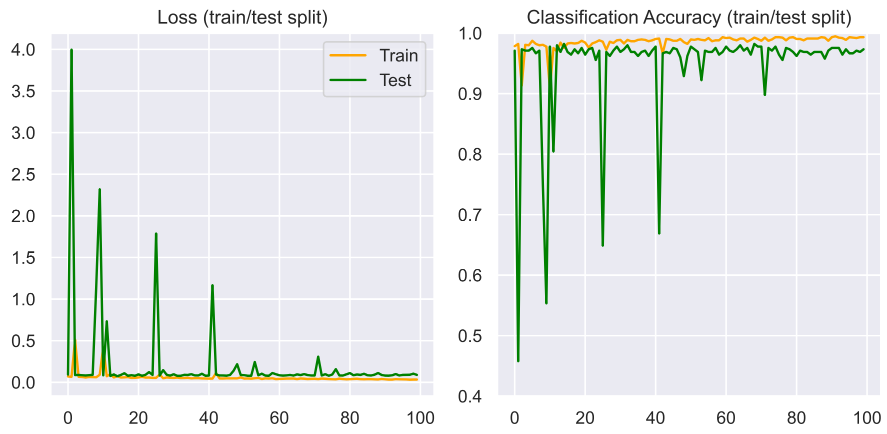


Figure 11: Loss and accuracy for a multi-layer NN, train/test split

## Convolutional neuron network

In this section, we illustrate a CNN with a smaller number of training parameters that is enough to show better performance then before.

To do so, we add a conv2D layer where convolutional layers learn filters that are set up as parameters. To reduce the number of parameters in a model, max pooling operation for 2D spatial data is used. To prevent overfitting and regularizing deep neural networks, dropout is implemented.

Let us look at the final model we chose, its structure is shown below. The average accuracy score on 5-fold cross validation achieves 0.992 and around 1.000 on the test set. This model has 32,764 training parameters, half as many as we used in a simple NN.

```
inputs_cnn=Input(shape=(8,8,1))
cnn_1=Conv2D(filters=4, padding='same',kernel_size=3, strides=(1,1),
    activation='relu')(inputs_cnn)
dr_1=Dropout(0.1)(cnn_1)
cnn_2=Conv2D(filters=8, padding='same',kernel_size=3, strides=(1,1),
    activation='relu')(dr_1)
mp_1=MaxPool2D(pool_size=2)(cnn_2)
dr_2=Dropout(0.1)(mp_1)

```

```
cnn_3=Conv2D(filters=16, padding='same',kernel_size=3, strides=(1,1),
    activation='relu')(dr_2)
dr_3=Dropout(0.1)(cnn_3)
fl=Flatten()(dr_3)
hidden_1=Dense(100, activation='relu', kernel_initializer='he_uniform')(fl)
hidden_2=Dense(50, activation='relu', kernel_initializer='he_uniform')(
    hidden_1)
output_cnn=Dense(10, activation='softmax')(hidden_2)
opt = Adam()
```

As it can be seen from Figure 13, the test loss converges to the train loss for all 5 folders as well as the accuracy score, thus it makes the model reliable enough to use.

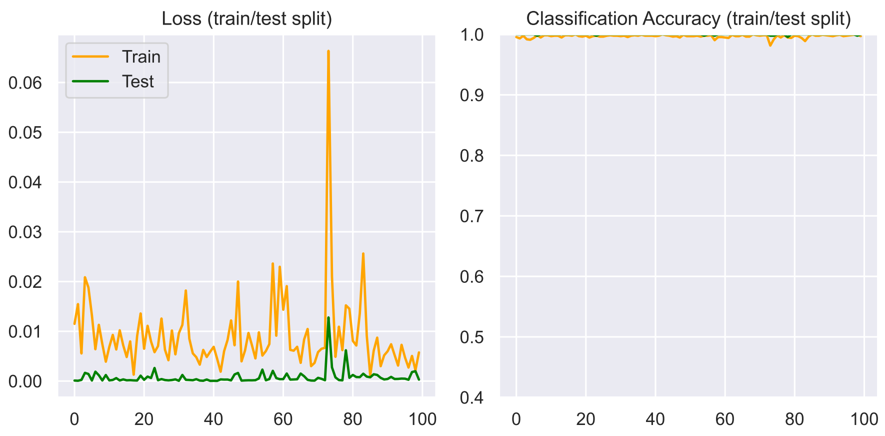


Figure 12: Loss and accuracy for a chosen CNN, train/test split

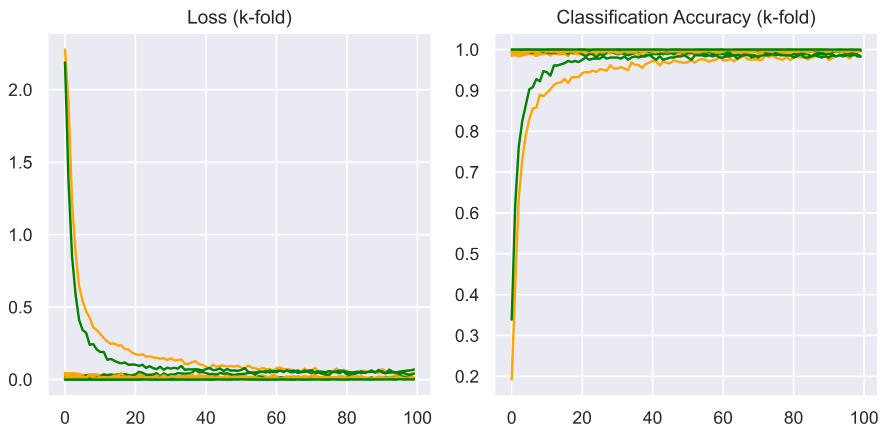


Figure 13: Loss and accuracy for a chosen CNN, 5-fold CV

## Splitting proportion

So far we worked with the data splitted into train and test sets with proportion 3:1. Here we study how this proportion results on the accuracy and loss functions. For six different split proportions taking values from 0.1 to 0.6, the accuracy for the chosen model is plotted on Figure 14.

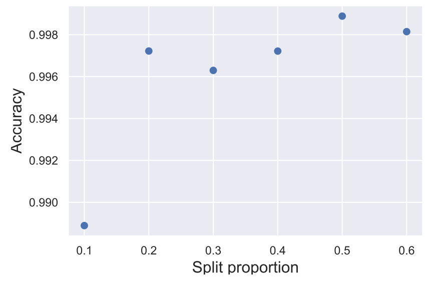


Figure 14: Accuracy for a chosen CNN for different proportion splits value

The minimum value of the accuracy of the test set is shown when a training set consists of 90% of the data, for other values of proportion the accuracy is relatively large. The first thought here is that training a model on a large data set while having a small one to validate the results can lead to overfitting. It should be noted that the results might be different for a data set with a higher resolution.

In the matter of the loss functions, as it was expected, having a small test set leads to overfit (large overfit gap on Figure 15 for 9:1 proportion).

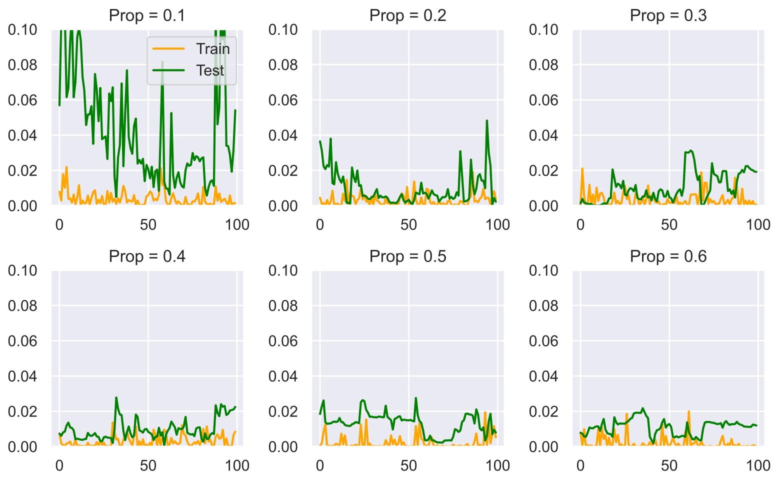


Figure 15: Loss for a chosen CNN for different proportion splits value

## 1.4 Summary

To summarise, three different methods have been applied to classify hand written digits form the MNIST dataset.

The stochastic subgradient projection method is suitable for large-scale problems with decomposition techniques. It is efficient in terms of processing time and precision of classification.

The Gaussian Process Classifier achieved a high precision in the classification, however the computational time increased significantly when the size of the training data increased.

In usual practice, convolutional neural networks achieve high performance as it was shown in this example. However, there are some cons to take into account. First, determining a proper network architecture is a challenging and time consuming process. Secondly, small amount of training data can lead to poor performance and thus some neural networks are pre-trained on different data sets. Also, there is a urgent problem with unpredictability and explainability of neural networks.

## 2 Appendix

### 2.1 Stochastic subgradient classifier

#### 2.1.1 Implementation

```
def loss_function(Theta, x, y):
# Returns the loss_value vector and the index j_star where j_star is the
    argmax of loss vector
    d, K = Theta.shape
    loss_vector = np.maximum(np.zeros([K, 1]), (np.ones([K, 1]) + x.
    transpose().dot(Theta - np.reshape(Theta[:,y], [-1, 1])).transpose()))
    loss_vector[y] = -1
    j_star = np.argmax(loss_vector)
    loss_value = loss_vector[j_star][0]
    return loss_value, j_star

def subgradient(Theta, x, y):
# Computes a subgrafient of the objective empirical hinge loss
# one pair of (x, y) given, x of size 1, and y an integer in {0, 1, ...,
    9}.
    g = np.zeros(Theta.shape)
    x = np.reshape(x, [-1, 1])
    y = int(y)
    d, K = Theta.shape
    loss, j_star = loss_function(Theta, x, y)
    if loss > 0:
        g[:, j_star] = x.transpose()
        g[:, y] = -x.transpose()
    return(g)

def svmsubgradient(Theta, x, y):
#  Returns a subgradient of the objective empirical hinge loss
# The inputs are Theta, of size n-by-K, where K is the number of classes,
# x of size n, and y an integer in {0, 1, ..., 9}.
    G = np.zeros(Theta.shape)
    for x_k, y_k in zip(x, y):
        G += subgradient(Theta, x_k, y_k)
    G /= x.shape[0]
    return(G)

def sgd(Xtrain, ytrain, maxiter = 10, init_stepsize = 1.0, l2_radius =
    10000, alpha = 0.5):
# Performs maxiter iterations of projected stochastic gradient descent
# on the data contained in the matrix Xtrain, of size n-by-d, where n
# is the sample size and d is the dimension, and the label vector
# ytrain of integers in {0, 1, ..., 9}. Returns two d-by-10
# classification matrices Theta and mean_Theta, where the first is the
    final
# point of SGD and the second is the mean of all the iterates of SGD.
# Each iteration consists of choosing a random index from n and the
# associated data point in X, taking a subgradient step for the
# multiclass SVM objective, and projecting onto the Euclidean ball
# The stepsize is init_stepsize / sqrt(iteration).
    K = 10
    NN, dd = Xtrain.shape
    print(NN)
    Theta = np.zeros(dd*K)
    Theta.shape = dd,K
    mean_Theta = np.zeros(dd*K)
    mean_Theta.shape = dd,K
    ## YOUR CODE HERE -- IMPLEMENT PROJECTED STOCHASTIC GRADIENT DESCENT
    for i in range(maxiter):
        index = np.random.randint(0, NN)
        stepsize = init_stepsize / ((i+1)**alpha)
        Theta -= stepsize * subgradient(Theta, Xtrain[index,:], ytrain[
    index])
        radius = np.linalg.norm(Theta.flatten())
        if radius > l2_radius:
            Theta = Theta / radius * l2_radius
        mean_Theta += Theta
    mean_Theta /= maxiter
    return Theta, mean_Theta
```

### 2.1.2 Evaluation

```
#choose a seed
seed = 1
np.random.seed(seed)
# Load data into train set and test set
digits = datasets.load_digits()
X = digits.data
y = np.array(digits.target, dtype = int)
X,y = shuffle(X,y)
N,d = X.shape
Ntest = np.int(100)

###### Question 1d : performance vs size of training sets
error_vector = np.zeros([6, 5])
Ntrains = [20, 50, 100, 500, 1000, 1500]
for i in range(len(Ntrains)):
    Xtrain = X[0:Ntrains[i],:]
    ytrain = y[0:Ntrains[i]]
    Xtest = X[Ntrains[i]:N,:]
    ytest = y[Ntrains[i]:N]

    alpha = 0.5
    l2_radius = 40.0
    M_raw = np.sqrt(np.mean(np.sum(np.square(Xtrain))))
    init_stepsize = l2_radius/M_raw
    maxiter = 40000
    for j in range(5):
        Theta, mean_Theta = sgd(Xtrain, ytrain, maxiter, init_stepsize,
    l2_radius, alpha)
        error_vector[i, j] = np.sum(np.not_equal(Classify(Xtest, mean_Theta
    ),ytest)/Ntest)

std = np.zeros([6, 1])
avg = np.zeros([6, 1])
for i in range(len(Ntrains)):
    std[i] = stdev(error_vector[i])
    avg[i] = mean(error_vector[i])

fig, ax = plt.subplots()
ax.errorbar(Ntrains, avg,
            yerr=std,
            fmt='-o')
ax.set_xlabel('Number of train samples')
ax.set_ylabel('Error')
ax.set_title('Errors of model depending on the number of training sets')
plt.show()

###### Question 1e : Performance vs learning rate
Ntrain = np.int(1697)
Xtrain = X[0:Ntrain,:]
ytrain = y[0:Ntrain]
Xtest = X[Ntrain:N,:]
ytest = y[Ntrain:N]

error_alpha_vector = np.zeros([10, 5])

alphas = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
for i in range(len(alphas)):
    alpha = alphas[i]
    l2_radius = 40.0
    M_raw = np.sqrt(np.mean(np.sum(np.square(Xtrain))))
    init_stepsize = l2_radius/M_raw
    maxiter = 40000
    for j in range(5):
        Theta, mean_Theta = sgd(Xtrain, ytrain, maxiter, init_stepsize,
    l2_radius, alpha)
        error_alpha_vector[i, j] = np.sum(np.not_equal(Classify(Xtest,
    mean_Theta),ytest)/Ntest)

std = np.zeros([10, 1])
avg = np.zeros([10, 1])
for i in range(len(alphas)):
    std[i] = stdev(error_alpha_vector[i])
    avg[i] = mean(error_alpha_vector[i])

```

```
fig, ax = plt.subplots()
ax.errorbar(alphas, avg,
            yerr=std,
            fmt='-o')
ax.set_xlabel('alpha')
ax.set_ylabel('Error')
ax.set_title('Errors of model depending on the stepsize parameter alpha')
plt.show()
```

## 2.2 Gaussian Process Classifier

```
"""
=====================================================
Gaussian process classification (GPC)
=====================================================

"""

import numpy as np
import matplotlib.pyplot as plt
import plotly.express as px
import pandas as pd
from sklearn import datasets
from sklearn.utils import shuffle
from sklearn.gaussian_process import GaussianProcessClassifier,
    GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, DotProduct,
    ConstantKernel, Matern, WhiteKernel, RationalQuadratic, ExpSineSquared
from sklearn.preprocessing import LabelBinarizer

def return_onehot(y):
    #Note, use LabelBinarizer instead of OneHotEncoder
    lb = LabelBinarizer()
    lb.fit(range(0,10,1))
    yonehot=lb.transform(y)
    #print(np.size(yonehot, 0),np.size(yonehot, 1) )
    return yonehot

def train_model_GPClass(Xtrain, ytrain, kernel, Xtest, ytest):
    #gpc_rbf = GaussianProcessClassifier(kernel=kernel).fit(Xtrain, ytrain)
    gpc_rbf = GaussianProcessClassifier(kernel=kernel, optimizer = None).
    fit(Xtrain, ytrain)
    #GaussianProcessClassifier performs hyperparameter estimation, this
    means that the the value specified above may not be the final
    hyperparameters
    #If you dont want it to do hyperparameter optimization set optimizer =
    None like this GaussianProcessClassifier(kernel=kernel,optimizer = None)
    .fit(Xtrain, ytrain)


    yp_train = gpc_rbf.predict(Xtrain)
    train_error_rate = np.mean(np.not_equal(yp_train,ytrain))

    yp_test = gpc_rbf.predict(Xtest)
    test_error_rate = np.mean(np.not_equal(yp_test,ytest))

    kernel_params = gpc_rbf.kernel_.get_params()['kernels']

    return train_error_rate, test_error_rate, str(kernel_params[0])

def train_model_GPRegr(Xtrain, ytrain, kernel, Xtest, ytest):
    #transform to onehot
    yonehot = return_onehot(ytrain)

    gpr = GaussianProcessRegressor(kernel=kernel).fit(Xtrain, yonehot)

    yp_train = gpr.predict(Xtrain)
    yp_train_indx = np.argmax(yp_train, axis = 1) #Find max of each row, i.
    e find class

    train_error_rate = np.mean(np.not_equal(yp_train_indx,ytrain))

    yp_test = gpr.predict(Xtest)
    yp_test_indx = np.argmax(yp_test, axis = 1) #Find class
    test_error_rate = np.mean(np.not_equal(yp_test_indx,ytest))

    #print(gpr.get_params())
    kernel_param = gpr.get_params()['kernel']

    return train_error_rate, test_error_rate, str(kernel_param)

def calculate_new_values(error_df, methods, training_sizes, kernels, iters
    = 3):

    # import data
    digits = datasets.load_digits()

    X = digits.data
    y = np.array(digits.target, dtype = int)
    N,d = X.shape

    for method in methods:
        for kernel_ind in range(len(kernels)):
            kernel = kernels[kernel_ind]

            for size_ind in range(len(training_sizes)):
                Ntrain = training_sizes[size_ind]
                temporary_df = error_df[(error_df['method'] == method) & (
    error_df['Ntrain'] == Ntrain) &(error_df['kernel'] == str(kernel) )]
                #Check that the value is not already calculated
```

```python
                if temporary_df.empty:

                    print('method = %s, Ntrain = %d, kernel = %s has not been prev. calc.' %(method, Ntrain,str(kernel)) )
                    try:
                        mean_test_error = 0
                        mean_train_error = 0
                        for i in range(iters): #do the errorcalculation multiple times, take average
                            X,y = shuffle(X,y)
                            Xtrain = X[0:Ntrain-1,:]
                            ytrain = y[0:Ntrain-1]
                            Xtest = X[Ntrain:N,:]
                            ytest = y[Ntrain:N]

                            if method == 'GPC':
                                train_error_rate, test_error_rate, kernel_params = train_model_GPClass(Xtrain, ytrain, kernel, Xtest, ytest) #a)
                            elif method == 'GPR':
                                train_error_rate, test_error_rate, kernel_params = train_model_GPRegr(Xtrain, ytrain, kernel, Xtest, ytest) #b)
                            else:
                                raise ValueError("%s is not an acceptable method" %method)

                            mean_train_error += train_error_rate / iters
                            mean_test_error += test_error_rate/iters
                        #      print(test_error_rate)
                        # print(mean_test_error)
                        print("Success when computing model")
                        row_df = {'method': method, 'Ntrain': Ntrain, 'kernel':kernel_params, 'train_error': mean_train_error , 'test_error': mean_test_error}
                        error_df = error_df.append(row_df, ignore_index=True)

                    except:
                        print("Error when computing model")
                else:
                    pass

    return error_df

def plot(error_df, methods, training_sizes, kernels):
    #Get the subdataframe to be plotted
    kernel_names = [str(kernels[i]) for i in range(len(kernels))]
    sub_df = error_df[(error_df['method'].isin(methods)) & (error_df['Ntrain'].isin(training_sizes)) &(error_df['kernel'].isin (kernel_names))
    ]

    #plot it
    fig = px.line(sub_df, x = 'Ntrain', y = 'test_error', color = 'kernel',
     line_dash = 'method', log_x = True, log_y = True)

    img_name = "images/" + "".join(methods) + str(kernels[0]) + ".png"
    fig.write_image(img_name)
    fig.show()
    return

def main():
    #choose a seed
    seed = 1
    np.random.seed(seed)

    training_sizes = [20, 50, 100, 500, 1000, 1500]
    #training_sizes = [20, 50, 100, 500]

    ##SPECIFY WHICH KERNELS TO USE
    kernels = []
    #kernels = [DotProduct(1.0), 1.0 * RBF([10]),  DotProduct() +
    WhiteKernel(10), Matern(7.5, nu = 0.5)];
    #kernels = [DotProduct() + ConstantKernel() +WhiteKernel(10), Matern(),
     RationalQuadratic(), ExpSineSquared()]

    params = [1,7.5,10,25,75]
    nu_vals = [ 0.5, 1.5, 2.5, 10]
    dot_params = [0,1e-5,1,10,100]
    for param in params:
        kernels.append(1.0*RBF(param))

    # for nu_val in nu_vals:
    #     kernels.append(Matern(7.5, nu = nu_val))
    # kernels.append(RBF(7.5))
    #
    # for param in dot_params:
    # #    kernels.append(DotProduct(param))
    #      kernels.append(DotProduct(10) + WhiteKernel(param))

    ##SPECIFY WHICH METHODS TO USE
    #methods = ['GPR']
    methods = ['GPR', 'GPC']

    ##USE DATAFRAME TO SAVE CALCULATIONS
    #error_df = pd.DataFrame(columns=['method', 'Ntrain', 'kernel', '
    train_error', 'test_error']) #Use if wanted to start from beginning
    error_df = pd.read_csv('error.csv', index_col = 0)

    error_df = calculate_new_values(error_df, methods, training_sizes,
    kernels)
    error_df.to_csv("error.csv")
    # print(error_df)

    plot(error_df, methods, training_sizes, kernels)

    return

main()
```

### 2.3 Evaluation functions for CNN

#### 2.3.1 Implementation

```
def evaluate(model, epochs, batch_size, X, y, n_folds=5):
  ''' Fuction for a model evaluation using 5-fold cross-validation '''
  scores, histories = [], []
  # Prepare cross-validation
  kfold = KFold(n_folds, shuffle=True, random_state=1)
  # Enumerate splits
  for train_ix, test_ix in kfold.split(X):
    # Select rows for train and test
    X_train, y_train, X_test, y_test = X[train_ix], y[train_ix], X[test_ix
   ], y[test_ix]
    # Fit model
    history = model.fit(X_train, y_train, epochs=epochs, batch_size=
   batch_size, validation_data=(X_test, y_test), verbose = 0)
    # Evaluate model
    _, acc = model.evaluate(X_test, y_test, verbose=0)
    # Store scores
    scores.append(acc)
    histories.append(history)
  return scores, histories


def diagnostics(fold_histories, histories):
  fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (8,4))
  ''' Diagnostics of the model learning behavior during training and the
   estimation of the model performance '''
  # Plotting a loss-finction
  ax1.set_title('Loss (train/test split)')
  ax1.plot(histories.history['loss'], color='orange', label='Train')
  ax1.plot(histories.history['val_loss'], color='green', label='Test')
  ax1.legend()
  # Plotting accuracy
  ax2.set_title('Classification Accuracy (train/test split)')
  ax2.plot(histories.history['accuracy'], color='orange')
  ax2.plot(histories.history['val_accuracy'], color='green')
  ax2.set_ylim(0.4, 1.)
  fig.tight_layout()
  # fig.savefig('finaltest.pdf')
  fig.show()

  plt.figure(figsize=(8,8))
  for i in range(len(fold_histories)):
    # Plotting a loss-finction
    plt.subplot(2, 2, 1)
    plt.title('Loss (k-fold)')
    plt.plot(fold_histories[i].history['loss'], color='orange', label='
   Train')
    plt.plot(fold_histories[i].history['val_loss'], color='green', label='
   Test')
    # Plotting accuracy
    plt.subplot(2, 2, 2)
    plt.title('Classification Accuracy (k-fold)')
    plt.plot(fold_histories[i].history['accuracy'], color='orange')
    plt.plot(fold_histories[i].history['val_accuracy'], color='green')
  plt.tight_layout()
  # fig.savefig('finaltestfold.pdf')
  plt.show()

  return

def scoreTest(kFold_accScores,acc_score):
  print('Average Accuracy Score on k-Fold Cross Validation:  %.3f' % (np.
   mean(kFold_accScores)))
  print('Accuracy Score on k-Fold Cross Validation:')
  for i in range(len(kFold_accScores)):
    print(f'{i+1}-Fold:  %.3f' % (kFold_accScores[i]))
  print('Accuracy Score on a Test Dataset:  %.3f' % (acc_score))
  return
```