EEE - 485 Term Project

Final Report

Muhammad Abdullah S. Mulkana - 21801075 Muhammad Ai Khan – 21801210

1. Introduction

One of the main tasks of Machine Learning is classification. Classification aims to group data based on certain features and can be applied to all sorts of dataset types, i.e., tabular, images, audio, etc. In our project, we aim to classify the body performance of people into four classes given certain physical features. The classification algorithms we employ are K Nearest Neighbors (KNN), Naïve Bayes, and Neural Networks. We experiment with different hyper-parameters and architectures in order to maximize the classification accuracy. We also use dimensionality reduction techniques such as Principal Component Analysis to understand how they affect our results.

2. Problem and Dataset Description

The task we are performing is multi-class classification. For this, we use a dataset for Body performance [1] available at Kaggle. The dataset contains 13393 observations, each consisting of 11 features related to human physiology which classify each person to one of four classes: A, B, C, D. Of these four classes, A indicates the best body performance and D indicates the worst. The features are a combination of continuous numerical features such as ‘weight’, discrete numerical features such as ‘situp counts’, and categorical features such as gender. After loading the dataset, we observe that the dataset contains no Null entries, therefore we do not need to cater to that issue.

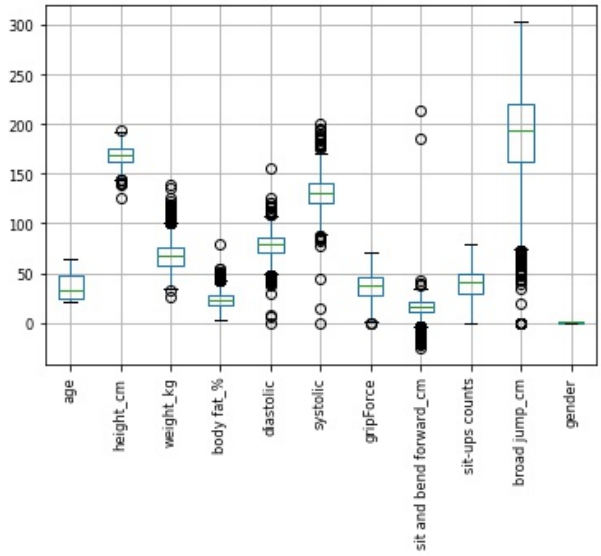


Figure 1: Boxplot of original data.

Figure 1 shows the boxplot of the features. The boxplot shows us the lower quartile (Q1) and upper quartile (Q3), median (Q2), minimum and maximum. This figure also shows us the outliers (as circles) which are defined as any point that is out of the following range: mean ± 1.5*IQR, where IQR = Q3-Q1. From this figure, we can see that the scale of the features vary where the median value for the 'broad jump_cm' is 193 and the median value for the 'body fat_%' is 22.8. This indicates the need to preprocess the data and rescale it so that no one feature overpowers the others for classifiers that are scale variant (such as KNN).

Figure 2 shows the histograms of the features and shows the distributions. Looking at the histogram of the class feature, we see that the bars are of the same height (A: 3348, B: 3347, C: 3349, D: 3349) meaning that we have a balanced dataset and we can use all the observations. The rest of the numerical features have an approximately Gaussian distribution ('age' has an exponential distribution).

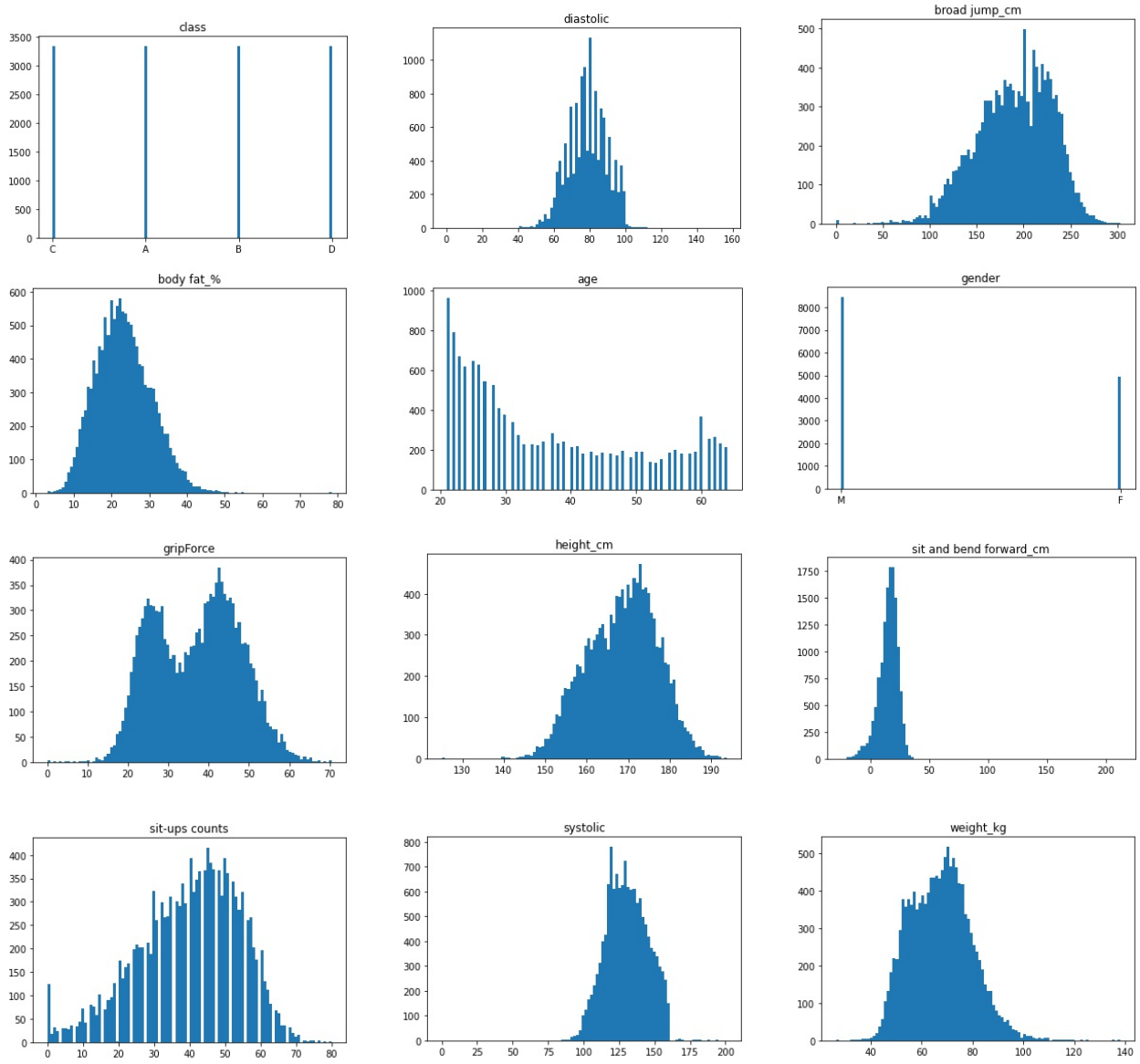
3. Methods

The following section outlines the methods we will employ during this project. Initially, we will preprocess our data so that we can use the classification models and obtain maximum accuracy. Next, we describe the classifiers that we will use.

3.1. Preprocessing

From the dataset description section, we know that there are no NULL values in the dataset. Next, we see that the only categorical feature that we have is gender which we

Figure 2: Histograms of the features.



preprocess by using one-hot-encoding [2]. Furthermore, we evade the dummy variable trap by dropping one of the columns [2]. Furthermore, we replace the categorical classes as follows: A: 1, B: 2, C: 3, D: 4. Next, we also observe from figures 1 and 2 that the scale of the features are very different. Therefore, we centralize and normalize the features by subtracting the means and dividing by the standard deviation. Lastly, we need to split the dataset into three parts: training, validation, and test. The training dataset is used for training of the models which will then be tested on the validation set. Since all models have hyperparameters that need tuning, we will iterate over possible hyperparameter values and test on the validation set. Then we choose the set of parameters where the validation set performs the best and we evaluate on the test set. This method will be applied to all the algorithms. We split the dataset into 80% training, 10% validation, and 10% testing. These three splits are kept the same over all models in order to accurately compare our results.

A technique that we will employ is Principal Component Analysis (PCA) to achieve dimensionality reduction. Currently we have 11 features but all of the features may not be important to describe the data. Therefore, we apply PCA to determine how much of the variance is captured by each dimension or Principal Component. Figure 3 (left) shows the cumulative explained variance ratio of the principal components. We see that we reach approximately 90% of explained variance with the first 6 PCs and the first 8 PCs capture 96.2%. We use the first 8 PCs in our classification models.

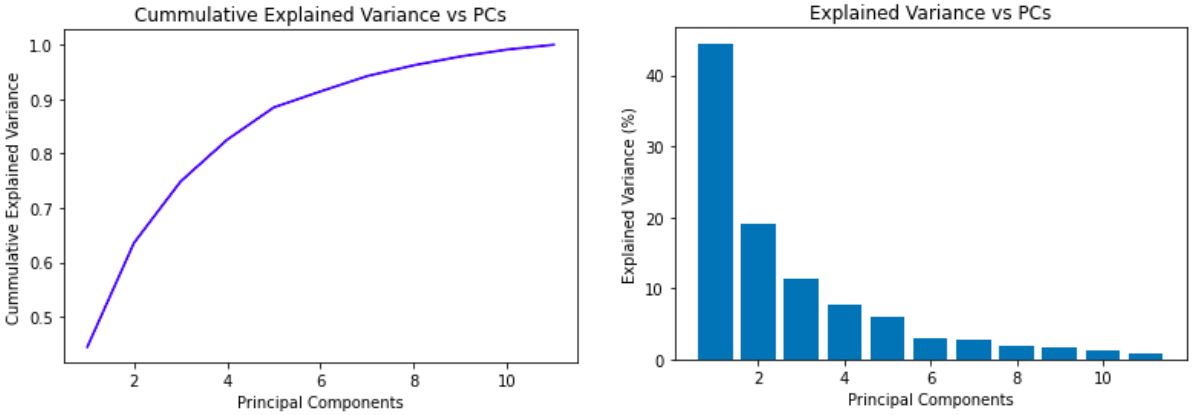


Figure 3: Cumulative and Individual Explained variances of all PCs.

Figure 3 (right) shows the individual variances captured by each PC. We can see that the first PC captures most of the variance i.e. approximately 45% while the second captures approximately 19%. The 6th PC and onwards capture less than 5% variance each.

3.2. K-Nearest Neighbors

K-Nearest Neighbors (KNN) is a classification technique where a certain observation is classified based on the labels of its neighbors. KNN is an example of a lazy learner [3]. These are models that have no training step and classify from scratch each time a test sample is presented. A certain distance metric is used to determine a sample's closest neighbors in the training dataset. The metrics we can use are Manhattan and Euclidean:

$$Euclidean = \sqrt{\sum_{i=1}^{N} (x_i - y_i)^2} \qquad Manhattan = \sqrt{\sum_{i=1}^{N} |x_i - y_i|}$$

Since we have already preprocessed our data by centralizing and standardizing the features, we know that all features will have the same importance. However, since we have outliers, we may see that the Manhattan distance performs better than the Euclidean distance. The KNN algorithm is as follows:

1. For each test sample:
    1. Calculate the distance to each training sample.

2. Get the labels of the K nearest neighbors.
3. Use majority voting to assign the label to the test sample. Majority voting means that the label with the highest frequency among the K nearest neighbor gets assigned to the test sample.

The hyper-parameters in this algorithm are K and the distance metric. Our evaluation metric is accuracy which is defined as the number of correct predictions divided by the total number of samples. We select the model with the highest accuracy on the validation set and then calculate the accuracy over the test set. KNN is chosen because it performs better on datasets where observations are more than the features when compared to other classifiers like SVM [4].

3.3. Naive Bayes

Naive Bayes is a classification technique that uses the Bayes Theorem. The one basic assumption it makes is that the features are independent with each other hence the name Naive Bayes [5]. Moreover, it is selected for this project as it is simple and fast to implement.

$$P(A \mid B) = (P(B \mid A).P(B))/(P(A))$$

The above equation is called the Bayes Rule and will be used in the Naive Bayes algorithm. In the above equation we will use A as our class vector such that $A = [1, 2, 3, 4]^T$ and B as our features vector such that $B = X = [X1, X2, \ldots, X11]^T$. In the above equation P(B|A) is a joint conditional probability, however, with the independent assumption, we can rewrite it as a product of individual probabilities. In the above equation, the denominator is disregarded as it does not impact the final outcome. We use this formula to find the $P(A|x_1, \ldots, x_{11})$ for all the classes and the maximum posterior probability is our prediction for the data instance. For each categorical data, we calculate its probability with respect to each class. For numerical data, we find the mean and variance of the feature and make an assumption that it is Gaussian and calculate the likelihood [6]. Looking at figure 2, we see that most of the features are Gaussian and the assumption of the features being Gaussian seems valid. We use the following formulas for this purpose:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma = \left[\frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \mu\right)^2\right]^{0.5}$$

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

The parameters to be calculated are the posterior probabilities for each class given the attributes, the likelihood probability and the prior probabilities for each class. The steps in Naive Bayes Classification are:

1. Calculate the prior probabilities for each class.
2. Find the likelihood probabilities for the given class labels.
3. Calculate the posterior probabilities using the above given formula.
4. Find the maximum probability given the input.

3.4. Feedforward Neural Network

Neural Networks are connected layers of neurons inspired by the human brain. We choose neural networks since they have the capacity to learn more about the data and to customize themselves to increase performance. Each layer in the network has a set of weights and an activation function. The weights are multiplied by the input which gives us the activation potential. The activation potential is passed through the activation function to get the output of the neurons in the layer. In a fully connected feedforward Neural Network, each of the previous neurons are connected to all the neurons in the following layer only, i.e. there is no connection that goes in the backward direction. The steps needed in developing a Feedforward Neural Network are: Initialization, Forward propagation, back-propagation. For our purposes, we will use p number of input neurons (p is the number of features). The number of hidden layers will be determined by validation as well as the number of neurons in each hidden layer. The output layer will consist of 4 neurons where each will output the probability of the sample belonging to a certain class.

To initialize the neural network weights and biases, we will use "Xavier initialization" [7] where for each hidden layer we initialize the weights and biases using a normal distribution with zero mean and variance $1/\mathrm{N}$ where N is the number of training samples [7]. This ensures that our weights are small enough so that they do not cause the exploding gradient problem which hinders learning. To train Neural Networks, we first need to perform forward propagation. For each layer, the activation potential can be calculated by multiplying that layer's input and the weight matrix. The activation functions that we will use in hidden layers can be one of the three:

$$Sigmoid: \phi(x) = \frac{1}{1+e^{-x}} \qquad tanh: \phi(x) = \tanh(x)$$
$$ReLU: \phi(x) = max(0, x)$$

Once we propagate to the output layer, we will use the softmax function which outputs the probabilities:

$$Softmax: \phi(x) = \frac{e^{x}}{\sum_{i=1}^{N} e^{x_i}}$$

The loss function we will use is categorical-cross entropy (CCE):

$$L = -\sum_{i=1}^{N} y_i * log\hat{y}_i$$

In order to calculate the loss, we need to convert our labels into one-hot-encoded versions. Since the SoftMax layer outputs probabilities that will be between 0 and 1, we can find which neuron gives the highest probability and then classify the sample to that class. Therefore, we have two performance metrics: accuracy and loss. To actually train the model, we will use mini-batch stochastic gradient descent where we will partition our training data into batches and train the model. For the back propagation, we will calculate the errors and gradients at each layer using the chain rule. In order to prevent overfitting, we can use regularization to force the model to reduce the magnitudes of the weights, thus increasing the ability of the model to generalize better. For our model, we will try the L_2 regularization to determine if it increases the performance.

$$L1\ regularization: L = CEE + \lambda\Sigma|W|$$

One challenge that we expect is to find the optimum structure for the neural network which we will achieve by means of validation.

4. Results

4.1 KNN

In the implementation of KNN for our data, we try two different distance metrics, i.e. Manhattan and Euclidean. Figure 4 (left) shows the accuracy on the validation set with respect to the k value when we run the model for k values between 1 and 50. We can see that the Manhattan distance produces better results. Figure 4 (right) shows the confusion matrix on the evaluation of the test data.

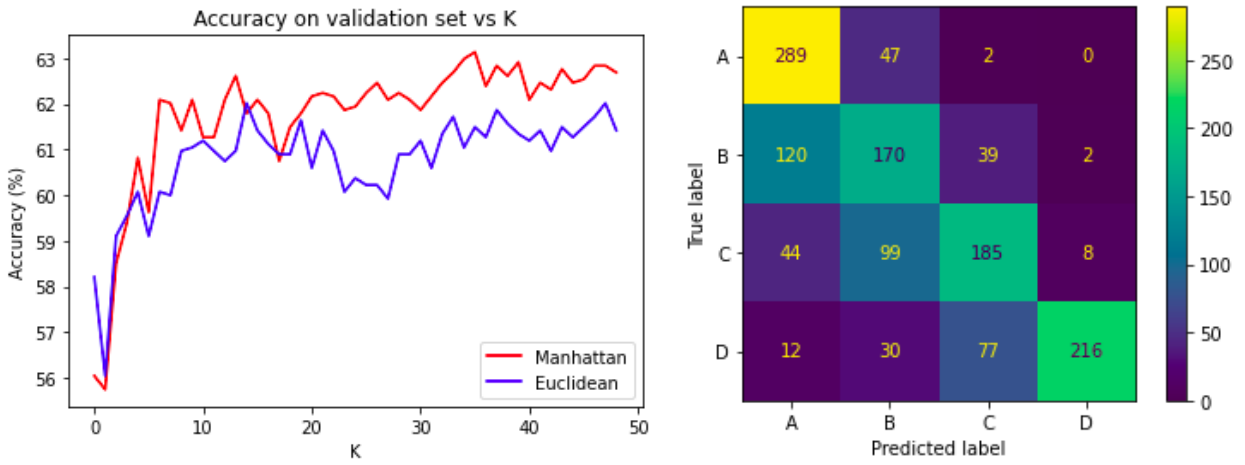


Figure 4: Accuracy plot and Confusion matrix for KNN on original data.

For the PCA data, as mentioned before, we choose the first 8 PCs as they capture approximately 96% of the variance. We expect our results to be not as accurate as we lose some information in the remaining 3 PCs that we ignore but we expect our model to run faster since we have a smaller dataset with only 8 features. Running the model for k values between 1 to 50, we get the following accuracy on the validation set:

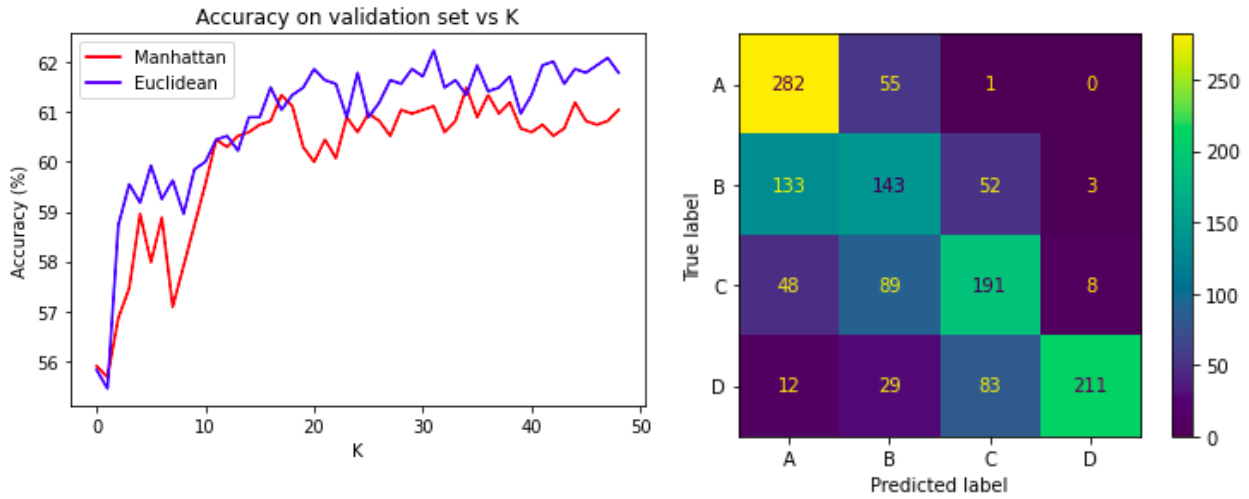


Figure 5: Accuracy plot and Confusion matrix for KNN on PCA reconstructed data.

As we can see from figure 5 (left) the Euclidean distance is the better distance metric when we have applied PCA on our data. Figure 5 (right) shows the confusion matrix of the best model.

Table 1 shows the summary of the results that we get from KNN. As we see, the best accuracy on both the validation and test is achieved when using the original dataset (without PCA) and when the distance metric is Manhattan. As expected, using PCA drops our accuracy but also drops the computational time (time for the model to run through 50 k values on validation set and once on the testing set)

Data	Max Validation Accuracy	Best K value	Test Accuracy	Distance Metric	Computational Time
Original	**63.13**	**36**	**64.18**	**Manhattan**	359.11 s
	62.01	15	62.23	Euclidean	
PCA	61.49	35	61.72	Manhattan	305.67
	62.24	32	62.76	Euclidean	

Table 1: Summary of KNN models.

4.2 Naïve Bayes

In the implementation of Naive Bayes, we used two different methods to evaluate our model. As stated in part 2, we have 10 numerical features and 1 categorical feature Gender. Hence, in the original model, we use all 11 features and in the second model, we remove the categorical feature 'gender' and use the remaining 10 numerical features. In our third model, we used the PCA to reduce our feature dimensions. Again, we use the top 8 PCs as they capture 96% of the variance. The accuracies and computational time for all three models are tabulated in Table 2 below:

Data	Validation	Feature	Computational
Original	52.84	11	0.38 s
Original w/o	52.16	10	0.43s
PCA	**55.30**	**8**	**0.23s**

Table 2: Summary of Naive Bayes models.

We can see in Table 2 that our model run on the PCA dataset is the best among the others with an accuracy of 56.12% on the validation set. PCA is used to combat the curse of dimensionality which states that as the number of features increases the accuracy increases until a certain point after which it starts to decrease [8]. The increase in accuracy for PCA is because higher dimensions capture noise and through dimensionality reduction, we are also removing this noise.

We evaluate the best performing model with the PCA transformed test data and the confusion matrix is shown below:

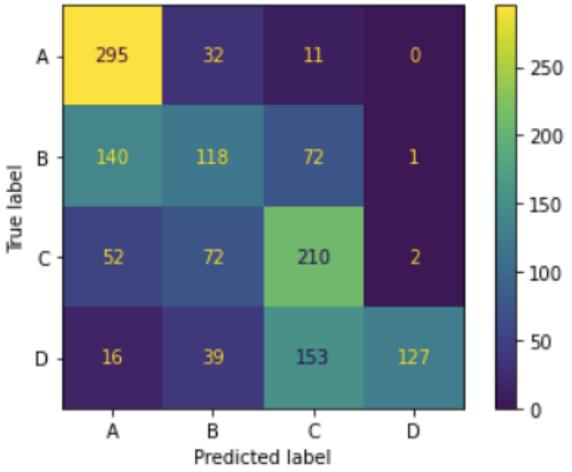


Figure 6: Confusion matrix for NB on PCA reconstructed data.

The classification accuracy on the test data is **55.9%.**

4.3 FNN

In the implementation of the Feed-Forward Neural Network, we experimented with different numbers of hidden layers and hyper parameters. The minimum number of hidden layers used was one whereas the maximum number was three. For each model, different numbers of neurons were used as well. Each hidden layer's activation function is the tanh function and the output layer's activation function is softmax where the loss was evaluated with categorical cross-entropy loss. We use mini-batch gradient descent with variable batch sizes and learning rates. Furthermore, the idea of momentum was used to increase performance. Table 3 summarizes the results of our experiments.

Table 3: Summary of Feed-Forward Neural Network models.

Configuration (Hidden Dimensions)	Data	lr	alpha	batches	Validation Accuracy	Epochs taken	Runtime
(50,20)	**origin**	**1e-4**	**0.99**	**128**	**70.0**	**181**	**21.25s**
	PCA				**69.1**	**173**	**19.2s**
(128,64,32)	origina	25e-5	0.99	1024	62.8	375	133s
	PCA				62.0	373	124s
(60,10)	origina	8e-5	0.97	256	67.1	801	89s
	PCA				67.0	2000	231s
(40)	origina	18e-	0.97	128	61.9	1600	126s

(40)	PCA	6	0.97	128	60.2	1616	127s

From the results above and other trials, we see that the optimal model has two hidden layers of 50 and 20 neurons respectively. Increasing the depth more does not increase performance. Furthermore, as stated in section 3.4, we planned on using regularization in order to combat any overfitting. However, in our trials we observed that there was very little overfitting and that regularization was affecting our performance in a negative way. The momentum coefficient 'alpha' is set to very high values, i.e. 0.97 and 0.99, which gave the best results. The main issue with our implementation is that it is prone to the exploding gradient problem and hence is very sensitive to the set of hyper parameters used. The best performing activation function for the hidden layers proved to be tanh which according to [7] works the best with our weight initialization method, i.e. "Xavier distribution."

Figure 7 shows the bar graph which visualizes the validation accuracies for models tested on both the original and PCA data.

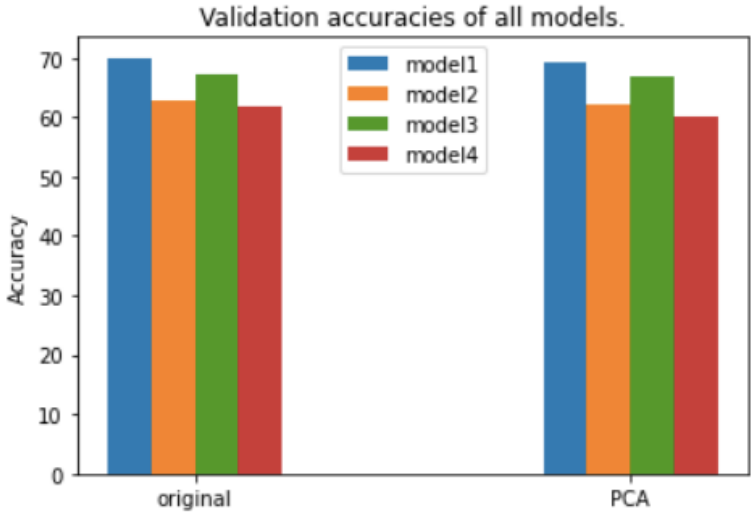


Figure 7: Bar Plot summarizing the validation accuracies of all models. (Model1, 2, 3, 4 refer to (50,20), (128, 64, 32), (60, 10), (40) respectively from table 3).

From table 3, we see that the best model is model 1. Figure 8 shows the training and validation accuracies during the training of this model.

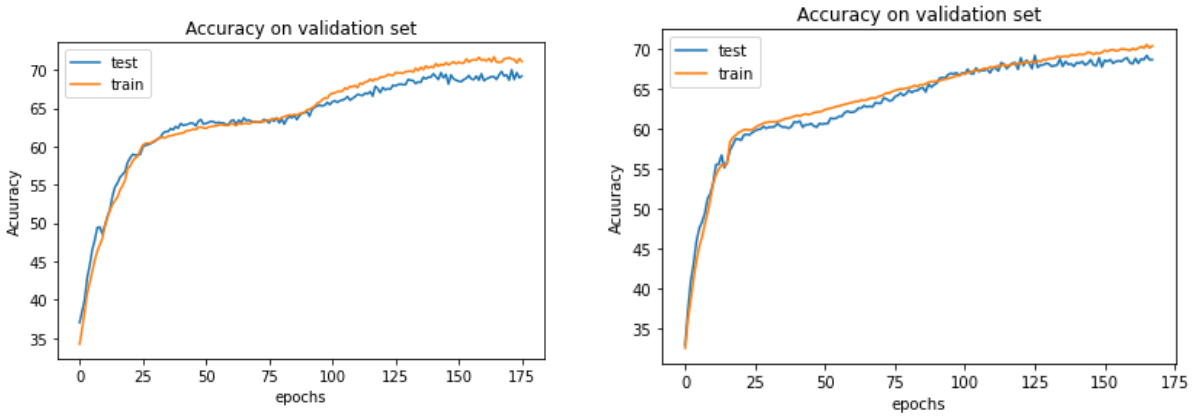


Figure 8: Training and Validation Accuracies for FNN models for original (left) and PCA (right) datasets.

After training, we evaluate our model using the test set. The test accuracy on the original test set was **68.1%** and on the PCA set, it was **65.4%**. Figure 9 shows the confusion matrix of the evaluation on the original and PCA test set.

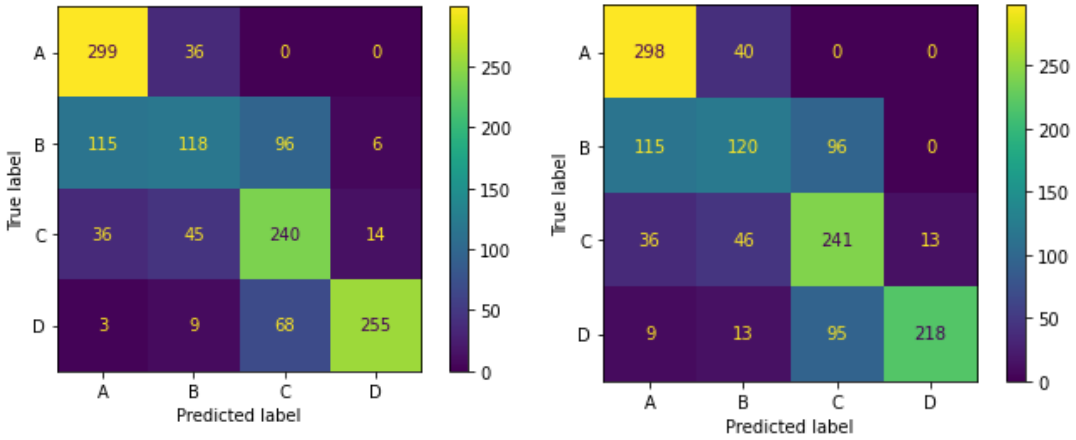


Figure 9: Confusion Matrix for FNN models for original (left) and PCA (right) datasets.

5. Conclusion

In this project our aim was to explore different classification algorithms and their configurations to determine the best performing model. In order to compare the configurations of each of the three classifiers, we use the validation set and then we evaluated the best models based on test set. Our first model is the k-nearest neighbor which gives the testing accuracy of 64.18% on the original data and 62.75% on the PCA datasets. Next, we implemented a Naïve Bayes Classification Model where the best model gave a testing accuracy of 56.12% on the PCA datasets as it had the best validation accuracy. Lastly, we implement a Feed-Forward Neural Network which is a more complex classification algorithm. After experimentation with numerous configurations, our best model gave a testing accuracy of 68.1% on the original and 65.4% on the PCA datasets. Comparing all three models based on classification accuracy, we can conclude that the best performing is the FNN this is because it has the capacity to learn more about the features by manipulating the network architecture.

Using PCA to transform the data helps in classifiers such as k-NN but the accuracy decreases when used with FNN. The advantage for using PCA with FNN is it decreases the training time. From our best performing model, we can see that the training time has decreased from 21.2s to 19.2s. Comparing the run time across all our models, we see that the Naïve Bayes takes the least amount of time due to its simple architecture. The reason for the relatively poor performance of the Naïve Bayes model can stem from the fact that the algorithm assumes independent features which may not be the case. For the KNN, the lack of performance may arise from the fact that the data is noisy and that the clusters for the four classes have some overlap. For the FNN, the major problem is the fact that the model lacks robustness and is extremely sensitive to the hyperparameters given and tends to suffer from the exploding gradient

problem. Overall, we can say that even though the FNN takes more time, it is the best model since it gives the best accuracy and the relative runtime is not significantly large.

6. References

[1] kukuroo3, "Body performance Data", kaggle (datasets), https://www.kaggle.com/kukuroo3/bodyperformance-data/version/12?select=bodyPerformance.csv

[2] K. K. Mahto, "One-hot-encoding, multicollinearity and the Dummy Variable Trap," *Medium*, 20-Jul-2019. [Online]. Available: https://towardsdatascience.com/one-hot-encoding-multicollinearity-and-the-dummy-variable-trap-b5840be3c41a.

[3] "KNN classification tutorial using Sklearn Python," *DataCamp Community*. [Online]. Available: https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn.

[4] D. Varghese, "Comparative Study on Classic Machine Learning Algorithms", *towards data science*, 6 th Dec 2018, https://towardsdatascience.com/comparative-study-on-classic-machinelearning-algorithms-24f9ff6ab222

[5] P. Vadapli, "Naive Bayes explained: Function, Advantages & disadvantages, applications in 2022," *upGrad blog*, 11-Jan-2022. [Online]. Available: https://www.upgrad.com/blog/naive-bayes-explained/#:~:text=Naive%20Bayes%20is%20suitable%20for,input%20variables%20than%20numerical%20variables.

[6] S. Chatterjee, "Use naive Bayes algorithm for categorical and Numerical Data Classification," *Medium*, 26-Nov-2020. [Online]. Available: https://medium.com/analytics-vidhya/use-naive-bayes-algorithm-for-categorical-and-numerical-data-classification-935d90ab273f.

[7] S. K. Kumar, "On weight initialization in deep neural networks, *CoRR*, *vol. abs/1704.08863*, 2017, Available: http://arxiv.org/abs/1704.08863

[8] A. Choudhury, "Curse of dimensionality and what beginners should do to overcome it," *Analytics India Magazine*, 09-Dec-2019. [Online]. Available: https://analyticsindiamag.com/curse-of-dimensionality-and-what-beginners-should-do-to-overcome-it/.

[9] M. A. S. Mulkana, M. A. Khan, "Term Project: First Report", Submitted on Moodle: 9th April. 2022.

7. Appendix

8.1 Exploration and split code

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv (r'bodyPerformance.csv')

''' Check for class imbalance '''

# classes = df[['class']].to_numpy()
classes = df[['class']]
# print(df.head())
# print(df.info())
# print(df.describe())
# print(df.skew())
# print(df.kurtosis())
unique, count = np.unique(classes, return_counts=True)
# print(unique)
# print(count)
samples = df.shape[0]

classes = classes.replace('A',1)
classes = classes.replace('B',2)
classes = classes.replace('C',3)
classes = classes.replace('D',4)

''' No class imbalance  '''
# cols = df.columns
# for col_name in cols:
   # hist = plt.hist(df[[col_name]], bins = 100)
   # plt.title(col_name)
   # plt.savefig('./figures/original-data/' + col_name + '-hist.jpg',bbox_inches = 'tight')
   # plt.show()


features = df.drop('class', axis = 1)
# print(features)
ohe = pd.get_dummies(features["gender"])
ohe_df = pd.get_dummies(features, columns=["gender"], drop_first=True)
ohe_df = ohe_df.rename(columns={"gender_M":"gender"})
# print(ohe_df.head())
# boxplot = ohe_df.boxplot(fontsize = 'small', rot = 90)
# boxplot_sns = sns.boxplot(df)

# plt.savefig('./figures/original-data/boxplot-org.jpg',bbox_inches = 'tight')


''' Center and Normalize '''
gender = ohe_df['gender']
```

```
mean = ohe_df[ohe_df.columns[:-1]].mean()
std = ohe_df[ohe_df.columns[:-1]].std()
features_center = (ohe_df - mean)/std
features_center['gender'] = gender

''' Test Train (Val??) Split'''
data = pd.concat([features_center, classes], axis=1)

test_size = 0.1*samples
validation_size = 0.1*samples
train_size = samples - test_size - validation_size

train_df = data.head(10713)
data_ntrain = data.tail(2680)
val_df = data_ntrain.head(1340)
test_df = data.tail(1340)

train_df.to_pickle('./data/train_data.pkl')
test_df.to_pickle('./data/test_data.pkl')
val_df.to_pickle('./data/val_data.pkl')
```

8.2 PCA Code

```
import numpy as np
import numpy.matlib
import time

import pandas as pd
import matplotlib.pyplot as plt

from sklearn.decomposition import PCA
from sklearn import neighbors
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

df = pd.read_csv (r'bodyPerformance.csv')

classes = df[['class']]

samples = df.shape[0]
k_show = 11
k = 8



classes = classes.replace('A',1)
classes = classes.replace('B',2)
classes = classes.replace('C',3)
classes = classes.replace('D',4)
features = df.drop('class', axis = 1)

ohe = pd.get_dummies(features["gender"])
```

```
ohe_df = pd.get_dummies(features, columns=["gender"], drop_first=True)
features = ohe_df.rename(columns={"gender_M":"gender"})

features = (features - features.mean())/features.std()

pca = PCA(n_components=k_show)
pca.fit(features)

pca.explained_variance_ratio_
exp_var = pca.explained_variance_ratio_
exp_var_cum = []
ind = []
var = 0
for i in range(len(exp_var)):
   ind.append(i+1)
   var = var + exp_var[i]
   exp_var_cum.append(var)
plt.plot(ind,exp_var_cum, 'b')
plt.title('Cummulative Explained Variance vs PCs')
plt.xlabel('Principal Components')
plt.ylabel('Cummulative Explained Variance')
plt.bar(ind,exp_var*100)
plt.title('Explained Variance vs PCs')
plt.xlabel('Principal Components')
plt.ylabel('Explained Variance (%)')
pca = PCA(n_components=k)
pca.fit(features)
data_sklearn = pca.fit_transform(features.to_numpy())
data = pd.concat([pd.DataFrame(data_sklearn), classes], axis=1)
train_df = data.head(10713)
data_ntrain = data.tail(2680)
val_df = data_ntrain.head(1340)
test_df = data.tail(1340)
train_df.to_pickle('./data/train_pca.pkl')
test_df.to_pickle('./data/test_pca.pkl')
val_df.to_pickle('./data/val_pca.pkl')
```

8.3 KNN Code

```
import numpy as np
import numpy.matlib
import pandas as pd
import time
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
start_time = time.time()
"""
------------------------------------------------------------------------
                        Functions
------------------------------------------------------------------------
"""
```

```
def predict(neighbor_list):
  neigh_labels = neighbor_list[1,:]
  lab, ins = np.unique(neigh_labels, return_counts=True)
  """maj vote"""
  lab_ind = np.argmax(ins)
  pred = lab[lab_ind]

  return pred


def accuracy(prediction, real):
  total = len(real)
  prediction = np.reshape(prediction,(-1,1))
  real = np.reshape(real,(-1,1))
  diff = prediction - real
  count = np.count_nonzero(diff)
  acc = (total - count)/total
  acc = acc*100
  return acc

def knn_opt(test, test_l, train_data, train_labels, k, p):
  num_test = np.shape(test)[0]
  num_train =  np.shape(train_data)[0]
  labels = []
  for i in range(num_test):
     test_sample = np.reshape(test[i,:], (1,-1))
     test_array = np.matlib.repmat(test_sample, num_train, 1)
     dist = np.sum(abs(np.subtract(test_array, train_data))**p,1)**(1/p) # Minkowski Distance
     dist = np.reshape(dist,(1,-1))
     train_labels1 = np.reshape(train_labels, (1,-1))
     dist_array1 = np.append(dist, train_labels1, axis = 0)
     dist_sort = dist_array1[:,dist_array1[0].argsort()]
     neighbors = dist_sort[:, 0:k]
     label = predict(neighbors)
     labels = np.append(labels, label)
  accur = accuracy(labels, test_l)
  return labels, accur



def load_data_pkl(path):
  df = pd.read_pickle(path)
  data = df.drop('class', axis = 1)
  labels = df[['class']]
  data = data.to_numpy()
  labels = labels.to_numpy()

  return data, labels


train_data, train_labels = load_data_pkl('./data/train_data.pkl')
```

```
test_data, test_labels = load_data_pkl('./data/test_data.pkl')
val_data, val_labels = load_data_pkl('./data/val_data.pkl')




num_feat =  np.shape(test_data)[1]
print(np.shape(test_data)[0])
print(np.shape(train_data)[0])
print(np.shape(val_data)[0])

p = 1 # Minkowski Distance. (1 - L1 (Best), 2 - Euclidean)

accur_manh_list = []
accur_euclidean_list = []
for k in range(1,50):
  labels_manh, accur_manh = knn_opt(val_data, val_labels, train_data, train_labels, k, 1)
  labels_euclidean, accur_euclidean = knn_opt(val_data, val_labels, train_data, train_labels, k, 2)

  print('K: ' + str(k))
  print('Validation Accuracy (Manhattan): ' + str(accur_manh))
  print('Validation Accuracy (Euclidean): ' + str(accur_euclidean))
  print("---------")
  accur_manh_list.append(accur_manh)
  accur_euclidean_list.append(accur_euclidean)

best_k_manh = np.argmax(accur_manh_list)+1
max_val_manh = np.max(accur_manh_list)
best_k_euclidean = np.argmax(accur_euclidean_list)+1
max_val_euclidean = np.max(accur_euclidean_list)


label1_manh, test_accur_manh = knn_opt(test_data, test_labels, train_data, train_labels, best_k_manh, 1)
label1_euclidean, test_accur_euclidean = knn_opt(test_data, test_labels, train_data, train_labels,
best_k_manh, 2)


cm = confusion_matrix(test_labels, label1_manh)
disp = ConfusionMatrixDisplay(cm, display_labels = ['A','B','C','D'])
disp.plot()
plt.show() # https://scikit-learn.org/stable/modules/generated/
sklearn.metrics.ConfusionMatrixDisplay.html#sklearn.metrics.ConfusionMatrixDisplay
print('Optimal K-value (Manhattan): ' + str(best_k_manh))
print('Maximum Validation Accuracy (Manhattan): ' + str(max_val_manh))
print('Test Accuracy (Manhattan): ' + str(test_accur_manh))

print('Optimal K-value (Euclidean): ' + str(best_k_euclidean))
print('Maximum Validation Accuracy (Euclidean): ' + str(max_val_euclidean))
print('Test Accuracy (Euclidean): ' + str(test_accur_euclidean))

plt.plot(accur_manh_list, 'r')
plt.plot(accur_euclidean_list, 'b')
```

```
plt.legend(['Manhattan', 'Euclidean'])
plt.title('Accuracy on validation set vs K')
plt.xlabel('K')
plt.ylabel('Accuracy (%)')
plt.show()
print("--- %s seconds ---" % (time.time() - start_time))
```

8.4 FNN Code

8.4.1 utils.py (classes Dense and FNN)

```
import numpy as np




''' This file contains the classes. Import this in the mail file. there are two classes. similar to tf, the hidden layer can be
created as an instance of 'Dense'. then create a list of dense layers and then call the FNN class.  '''

class FNN:

  def __init__(self, layers_list):
    self.layers = layers_list
    self.act_fncs = []
    self.test_error = []
    self.test_acc = []
    self.train_error = []
    self.train_acc = []
    self.best_acc = -10


  def forward(self, x):
    h = x
    for l in self.layers:
      l.input = h
      act_pot = np.dot(h, l.w) + l.b
      h = l.activation_function(act_pot,l.act_func)
      l.output = h
      l.v = act_pot
    return h



  def cee(self, true, pred):
    error = 0
    pred = np.clip(pred, 0.0000001, 1-0.0000001)
    # The clip is done to avoid any issues in the CCE error. this idea is inspired form https://stackoverflow.com/questions/65131391/what-exactly-is-kerass-categoricalcrossentropy-doing
    ln_val = np.log(pred)
    t1 = ln_val*true
    error = (-1 * np.sum(t1))/(2*true.shape[0])
```

```
  return error + self.reg_param*self.reg_sq() /(2*true.shape[0])



 def accuracy(self, true, pred):
  pred_labels = np.argmax(pred, axis = 1)
  true_labels = np.argmax(true, axis = 1)
  accurate = sum(pred_labels == true_labels)*100
  return (accurate/true.shape[0]), pred_labels



 def reg_sq(self):
    sum_weights = 0
    for l in self.layers:
       sum_weights += sum(sum(l.w*l.w))
    return sum_weights


 def backward(self, y):

     for l in reversed(self.layers):
        if l.act_func == 'softmax':
           l.delta =  (l.output- y)
           l.grad = np.matmul(l.input.T,l.delta) + (self.reg_param*l.w) /(y.shape[0])

        else:
           der = l.der_activation_function()
           l.delta = np.multiply(der,error)
           l.grad = np.matmul(l.input.T,l.delta)+ (self.reg_param*l.w) /(y.shape[0])

        l.grad_b = np.sum(l.delta, axis = 0)
        upd_w = self.lr * (l.grad/self.batch_size ) + self.alpa*l.upd_w
        l.upd_w = upd_w
        upd_b = self.lr * (l.grad_b/self.batch_size ) + self.alpa*l.upd_b
        l.upd_b = upd_b
        error = np.matmul(l.delta,l.w.T)




 def update(self):
  for l in self.layers:
    l.w = l.w - l.upd_w
    l.b = l.b - l.upd_b



 def train(self, tr_data, tr_label, v_data, v_label, te_data, te_labels, epochs, batch_size, lr, reg_param,
alpha):
   batches = np.floor(tr_data.shape[0]/batch_size)
```

```
    self.lr = lr
    self.batch_size = batch_size
    self.reg_param = reg_param
    self.alpa = alpha


    prev_err = 0
    count = 0
    flag = True
    epoch = 0


    while flag == True and epoch < epochs:

      rand_ind = np.random.permutation(tr_data.shape[0])
      input_data = tr_data[rand_ind]
      output = tr_label[rand_ind]

      for batch in range(int(batches)):
        l_ind =  int(batch_size*batch)
        u_ind = int(batch_size*batch+batch_size)
        x = input_data[l_ind:u_ind]
        y = output[l_ind:u_ind]

        self.forward(x)
        self.backward(y)
        self.update()

      e, a, v_preds = self.evaluate(v_data, v_label)
      e_t, a_t, tr_preds = self.evaluate(tr_data, tr_label)
      self.test_error.append(e)
      self.test_acc.append(a)
      self.train_error.append(e_t)
      self.train_acc.append(a_t)
      print('-------------')
      print('Epoch: ' + str(epoch) + ' Validation loss: ' + str(e) + ' Training loss: ' + str(e_t)+ ' Validation acc: '
+ str(a)+ ' Training acc: ' + str(a_t))
      self.grad = []
      self.delta = []

      if a > self.best_acc:
          self.t_e, self.t_a, self.te_preds = self.evaluate(te_data, te_labels)
          self.best_acc = a


      if e - prev_err > 0.2 :
          count +=1

      if count == 2 or np.isnan(e):
          flag = False
      prev_err = e
      epoch +=1
```

```
  def evaluate(self, data, labels):
    output = self.forward(data)
    error = self.cee(labels,output)
    acc, preds = self.accuracy(labels, output)
    return error, acc, preds


class Dense:
    def __init__(self,input_neurons, output_neurons, act_func, init_method, sigma):
        self.input_neurons = input_neurons
        self.output_neurons = output_neurons
        self.act_func = act_func
        if init_method == 'xavier': #previous xavier init method. doesnt work as good as the newer Xavier
method as stated in report
            w_0 = ((6/(input_neurons+output_neurons))**0.5)/10
            self.w = np.random.uniform(-1*w_0,w_0,[input_neurons,output_neurons])
            self.b = np.random.uniform(-1*w_0,w_0,[1,output_neurons])

        elif init_method == 'normal':
            self.w = np.random.normal(sigma, size=(input_neurons,  output_neurons))
            self.b = np.random.normal(sigma, size=(1,  output_neurons))
        else:
            print('invalid method')
        self.upd_w = np.zeros(self.w.shape)
        self.upd_b = np.zeros(self.b.shape)




    def activation_function(self, x, function):
      if function == 'sigmoid':
        y = np.exp(x)/(1+np.exp(x))
      elif function == 'tanh':
        y = np.tanh(x)
      elif function == 'relu':
        y = np.maximum(0.,x)
      elif function == 'softmax':
        y = np.exp(x)/np.sum(np.exp(x), axis=0)
      else:
        print('Activation function not valid')
      return y



    def der_activation_function(self):
      if self.act_func == 'sigmoid':
        y = np.multiply(self.output,(1-self.output))
      elif self.act_func == 'tanh':
          y = (1-self.output**2)
      elif self.act_func == 'relu':
```

```
      potential = self.v
      potential[potential<=0] = 0.0
      potential[potential>0] = 1.0
      y = potential
    else:
      print('Activation function not valid')
    return y
```

8.4.2 training FNN

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 20 13:19:29 2022

@author: muhammadabdullah
"""



import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
from utils import Dense, FNN
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay



def load_data_pkl(path):
    df = pd.read_pickle(path)
    data = df.drop('class', axis = 1)
    labels = df[['class']]
    data = data.to_numpy()
    labels = labels.to_numpy()

    return data, labels


def one_hot_encode(y):
  ohe = np.zeros([y.shape[0],4])
  for i in range(len(y)):
    ohe[i,(y[i]-1)] = 1
  return ohe


''' LOAD DATA'''
train_data, train_labels = load_data_pkl('./data/train_pca.pkl')
test_data, test_labels = load_data_pkl('./data/test_pca.pkl')
val_data, val_labels = load_data_pkl('./data/val_pca.pkl')


''' OHE'''
```

```
val_labels = one_hot_encode(val_labels)
train_labels = one_hot_encode(train_labels)
test_labels = one_hot_encode(test_labels)


''' Dataset parameters'''

num_feat =  np.shape(test_data)[1]
train_size = np.shape(train_data)[0]
test_size = np.shape(test_data)[0]
val_size = np.shape(val_data)[0]




start = time.time()

''' INIT LAYERS'''

layers = [Dense(8,128,'tanh','normal', 1/train_size),
          Dense(128,64,'tanh','normal', 1/train_size),
          Dense(64,32,'tanh','normal', 1/train_size),
          Dense(32,4,'softmax','normal', 1/train_size)]

''' INIT MODEL'''

model = FNN(layers)

''' MODEL PARAMETERS'''

epochs = 2000
batch_size = 1024
lr =  0.00025
l = 0.00
alpha = 0.99


''' RUN MODEL'''

model.train(train_data, train_labels,  val_data,  val_labels, test_data, test_labels, epochs, batch_size, lr, l,
alpha)
print('runtime {}s'.format(time.time()-start))

''' plot MODEL results'''
plt.plot(model.train_error)
plt.show()
plt.plot(model.test_error)
plt.show()
plt.plot(model.test_acc, label = 'test')
plt.plot(model.train_acc, label = 'train')
```

```
plt.legend()
plt.show()

true_labels = np.argmax(test_labels, axis = 1)
cm = confusion_matrix(true_labels, model.te_preds)
disp = ConfusionMatrixDisplay(cm, display_labels = ['A','B','C','D'])
disp.plot()
plt.show()


''' script for the barplots'''
# model1 = [70,69.1]
# model2 = [62.8,62]
# model3 = [67.1,67]
# model4 = [61.9,60.2]

# width = 0.1
# md1 = np.arange(len(model1))
# md2 = [x + width for x in md1]
# md3 = [x + width for x in md2]
# md4 = [x + width for x in md3]



# # creating the bar plot
# plt.bar(md1,model1, width = barWidth, label ='model1')
# plt.bar(md2, model2, width = barWidth, label ='model2')
# plt.bar(md3,model3, width = barWidth, label ='model3')
# plt.bar(md4,model4, width = barWidth, label ='model4')
# plt.xticks([r + 1.5*width for r in range(len(model1))],
#         ['original', 'PCA'])
# plt.legend(loc=9)
# plt.title('Validation accuracies of all models.')
# plt.ylabel('Accuracy')
# plt.savefig('barplot.png', dpi=200)
# plt.show()

# for the barplot parameters and code, we used change code from https://www.geeksforgeeks.org/bar-plot-in-matplotlib/
```

8.5 Naïve Bayes Codes

8.5.1 Naïve Bayes

```
import numpy as np
import pandas as pd
import time
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
```

```
def load_data_pkl(path):
    df = pd.read_pickle(path)
    data = df.drop('class', axis = 1)
    labels = df[['class']]
    data = data.to_numpy()
    labels = labels.to_numpy()

    return data, labels


def NBclassifier(prior, X_train, Y_train, X_val, Y_val):
    cat = X_train[:,4]
    num = np.delete(X_train,4,1)
    data_stats = dict()
    for i in range(1,5):
        ind_class = np.where(Y_train == i)[0]
        cat_class = cat[ind_class]
        num_class = num[ind_class]

        Pm_class = sum(cat_class) / len(ind_class)
        Pf_class = sum(1-cat_class) / len(ind_class)
        class_mu = np.mean(num_class,axis =0)
        class_var = np.var(num_class, axis=0)
        sigma = np.diag(class_var)

        data_stats[str(i)+'_mu'] = class_mu
        data_stats[str(i)+'_var'] = sigma
        data_stats[str(i)+'_male'] = Pm_class
        data_stats[str(i)+'_female'] = Pf_class
        data_stats[str(i)+'_coef'] = 1/(((2*np.pi)**(num_class.shape[1]/2)) * abs((np.linalg.det(sigma)))*0.5)


    pred = []
    count = 0
    cat = X_val[:,4]
    num = np.delete(X_val,4,1)
    for i in range(len(X_val)):
        test_sample = num[i]
        test_label = Y_val[i]
        prob_list = []

        for j in range(4):
            mu = data_stats[str(j+1)+'_mu']
            cov = data_stats[str(j+1)+'_var']
            Male = data_stats[str(j+1)+'_male']
            Female = data_stats[str(j+1)+'_female']
            coef = data_stats[str(j+1)+'_coef']

            dif = np.transpose(test_sample-mu)
            if cat[i] == 1:
                temp = Male
            else:
                temp = Female
```

```
            prob = temp*prior[j]*coef*np.exp(-0.5*np.matmul(np.matmul(dif, np.linalg.inv(cov)), dif))
            prob_list.append(prob)
        prediction = np.argmax(prob_list)+1
        pred.append(prediction)

        if prediction == test_label:
            count = count + 1
    accuracy = count/X_val.shape[0]

    return accuracy, pred
    # return data_stats

start = time.time()
# Load the data
train_data, train_labels = load_data_pkl('./data/train_pca.pkl')
test_data, test_labels = load_data_pkl('./data/test_pca.pkl')
val_data, val_labels = load_data_pkl('./data/val_pca.pkl')

classes, count = np.unique(train_labels, return_counts = True)
prior = count/train_data.shape[0]


t, nb_pred = NBclassifier(prior, train_data, train_labels, val_data, val_labels)
cm = confusion_matrix(test_labels, nb_pred)
disp = ConfusionMatrixDisplay(cm, display_labels = ['A','B','C','D'])
disp.plot()
plt.show()
print(t*100)
print('runtime {}s'.format(time.time()-start))
```

8.5.2 Naïve Bayes with no categorical feature

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Apr 30 16:24:58 2022

@author: muhammadabdullah
"""

''' NB W/O'Categorical '''

import numpy as np
import pandas as pd
import time
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt




def load_data_pkl(path):
    df = pd.read_pickle(path)
```

```
    data = df.drop('class', axis = 1)
    labels = df[['class']]
    data = data.to_numpy()
    labels = labels.to_numpy()

    return data, labels


def NBclassifier(prior, X_train, Y_train, X_val, Y_val):
    data_stats = dict()
    X_train = np.delete(X_train,4,1)
    for i in range(1,5):
        ind_class = np.where(Y_train == i)[0]
        num_class = X_train[ind_class]

        class_mu = np.mean(num_class,axis =0)
        class_var = np.var(num_class, axis=0)
        sigma = np.diag(class_var)

        data_stats[str(i)+'_mu'] = class_mu
        data_stats[str(i)+'_var'] = sigma
        data_stats[str(i)+'_coef'] = 1/(((2*np.pi)**(num_class.shape[1]/2)) * abs((np.linalg.det(sigma)))*0.5)


    pred = []
    count = 0
    X_val = np.delete(X_val,4,1)
    for i in range(len(X_val)):
        test_sample = X_val[i]
        test_label = Y_val[i]
        prob_list = []

        for j in range(4):
            mu = data_stats[str(j+1)+'_mu']
            cov = data_stats[str(j+1)+'_var']
            coef = data_stats[str(j+1)+'_coef']
            dif = np.transpose(test_sample-mu)
            prob = prior[j]*coef*np.exp(-0.5*np.matmul(np.matmul(dif, np.linalg.inv(cov)), dif))
            prob_list.append(prob)
        prediction = np.argmax(prob_list)+1
        pred.append(prediction)

        if prediction == test_label:
            count = count + 1
    accuracy = count/X_val.shape[0]

    return accuracy, pred
start = time.time()

# Load the data
train_data, train_labels = load_data_pkl('./data/train_data.pkl')
test_data, test_labels = load_data_pkl('./data/test_data.pkl')
val_data, val_labels = load_data_pkl('./data/val_data.pkl')
```

```
classes, count = np.unique(train_labels, return_counts = True)
prior = count/train_data.shape[0]
t, nb_pred = NBclassifier(prior, train_data, train_labels, val_data, val_labels)
cm = confusion_matrix(test_labels, nb_pred)
disp = ConfusionMatrixDisplay(cm, display_labels = ['A','B','C','D'])
disp.plot()
plt.show()
print(t*100)
print('runtime {}s'.format(time.time()-start))
```

8.5.3 Naïve Bayes with PCA

```
''' NB PCA'''

import numpy as np
import pandas as pd
import time
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt


def load_data_pkl(path):
    df = pd.read_pickle(path)
    data = df.drop('class', axis = 1)
    labels = df[['class']]
    data = data.to_numpy()
    labels = labels.to_numpy()

    return data, labels


def NBclassifier(prior, X_train, Y_train, X_val, Y_val):
    data_stats = dict()
    for i in range(1,5):
        ind_class = np.where(Y_train == i)[0]
        num_class = X_train[ind_class]

        class_mu = np.mean(num_class,axis =0)
        class_var = np.var(num_class, axis=0)
        sigma = np.diag(class_var)

        data_stats[str(i)+'_mu'] = class_mu
        data_stats[str(i)+'_var'] = sigma
        data_stats[str(i)+'_coef'] = 1/(((2*np.pi)**(num_class.shape[1]/2)) * abs((np.linalg.det(sigma)))*0.5)


    pred = []
    count = 0
    for i in range(len(X_val)):
        test_sample = X_val[i]
        test_label = Y_val[i]
        prob_list = []
```

```
        for j in range(4):
            mu = data_stats[str(j+1)+'_mu']
            cov = data_stats[str(j+1)+'_var']
            coef = data_stats[str(j+1)+'_coef']

            dif = np.transpose(test_sample-mu)

            prob = prior[j]*coef*np.exp(-0.5*np.matmul(np.matmul(dif, np.linalg.inv(cov)), dif))
            prob_list.append(prob)
        prediction = np.argmax(prob_list)+1
        pred.append(prediction)

        if prediction == test_label:
            count = count + 1
    accuracy = count/X_val.shape[0]

    return accuracy, pred
    # return data_stats



start = time.time()
# Load the data
train_data, train_labels = load_data_pkl('./data/train_pca.pkl')
test_data, test_labels = load_data_pkl('./data/test_pca.pkl')
val_data, val_labels = load_data_pkl('./data/val_pca.pkl')

classes, count = np.unique(train_labels, return_counts = True)
prior = count/train_data.shape[0]


t, nb_pred = NBclassifier(prior, train_data, train_labels, val_data, val_labels)
cm = confusion_matrix(test_labels, nb_pred)
disp = ConfusionMatrixDisplay(cm, display_labels = ['A','B','C','D'])
disp.plot()
plt.show()
print(t*100)
print('runtime {}s'.format(time.time()-start)
```