CSE315 Assignment 2

MNIST Classification
SVM and PCA
K-Means Clustering

Name: Wilbert Osmond
Student ID: 1926308

Major: Exchange (non-UoL)
University: Xi'an Jiaotong-Liverpool University (XJTLU)
Region: Suzhou, Jiangsu, China Mainland

Period: Fall 2019
Date: 14/12/2019 (mitigating circumstances)

Contents

Overview

Throughout the following project, different classification algorithms implemented on MNIST and iris datasets with some visualizations were explored and implemented in Python code. This project unites our lecture work with that of our laboratory tutorials, showcasing our skills in both supervised and unsupervised algorithms. Through Python coding of CNN, SVM and K-Means clustering, this project allows us for a full understanding of the step-by-step of each concept. This project displays the importance of the relevant concepts, proper Python coding and improves critical analysis skills of the students completing it.

Note that the Python codes are written in iPython Notebook. It is also important to note that packages that have been imported previously may not be showed to be imported again in the next figures of code.

Task 1: Classification [35 Marks]

1.1. Using the MNIST database available at http://yann.lecun.com/exdb/mnist/, select two classification algorithms and implement them to achieve a high accuracy (more than 90%). [15 marks]

Data preparation that is supposed to precede these two algorithms will be shown and discussed later in task 1.2. The most relevant thing for now to note from the data preparation is that *x* is the pixel array and *y* is the target array.

Classification algorithm 1 – Convolutional Neural Network (CNN)

We use the Keras Sequential API, which allows us to easily stack sequential layers of the network in order by adding one layer at a time, starting from the input, as shown in Figure 1.1.1. I start by declaring the model type as *Sequential().*

The first is the 2D convolutional (Conv2D) layer to process the 2D MNIST input images. It is similar to a set of learnable filters. The first argument passed to the *Conv2D()* layer function is the number of output channels – in this case we have 28 output channels. The next input is the *kernel_size*, which in this case we have chosen to be a 3x3 moving window. Each filter transforms a part of the image (defined by the kernel size) using the kernel filter. The kernel filter matrix is applied on the whole image. Filters can be seen as a transformation of the image. Next, the activation function is a rectified linear unit (activation function *max(0, x)*. The rectifier activation function is used to add non linearity to the network. Finally, we have to supply the model with the size of the input to the layer. Declaring the input shape is only required of the first layer – Keras is good enough to work out the size of the tensors flowing through the model from there. Also notice that we don't have to declare any weights or bias variables as Keras sorts that out for us. The CNN can isolate features that are useful everywhere from these transformed images (feature maps).

The second important layer in CNN is the pooling (*MaxPool2D*) layer. This layer simply acts as a downsampling filter. As we specify the stride to be 2x2, it looks at the 2 neighbouring pixels and

picks the maximal value. These are used to reduce computational cost, and to some extent also reduce overfitting. We have to choose the pooling size (i.e., the area size pooled each time) more the pooling dimension is high, more the downsampling is important.

Combining convolutional and pooling layers, the CNN is able to combine local features and learn more global features of the image.

The flatten layer is used to convert the final feature maps into a one single 1D vector. This flattening step is needed so that we can make use of fully connected layers after some convolutional/maxpool layers. It combines all the found local features of the previous convolutional layers.

In the end, we use the features in two fully-connected (*Dense*) layers which is just an artificial neural networks (ANN) classifier with a dropout layer in the middle. The first dense layer is a fully connected layer with relu activation with 128 outputs. The dropout layer is a regularization method which will be discussed in subtask 1.3. In the last layer, (*Dense(10, activation="softmax")*) the net outputs distribution of probability of each class of the total 10 classes.

```
In [30]: from tensorflow.keras.models import Sequential
         from tensorflow.keras.layers import Input
         from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
         from tensorflow.keras.layers import Conv2D, MaxPooling2D

         # Prepare a model (sequential) and adding layers accordingly data shape from .shape
         train_model = Sequential()
         train_model.add(Conv2D(28, kernel_size=(3,3), activation='relu', input_shape=input_shape))
         train_model.add(MaxPooling2D(pool_size=(2,2)))
         train_model.add(Flatten()) # flatten to FC layers
         train_model.add(Dense(128, activation=tf.nn.relu)) # fully connected layer with relu activation
         train_model.add(Dropout(0.2))
         train_model.add(Dense(10,activation=tf.nn.softmax)) # fully-conected layer for output with softmax classification, 10 classes
```

Figure 1.1.1: Defining the CNN model by specifying the layers.

After defining the CNN model, now we train our CNN model. Before that we need to compile the model in advance. For the loss function, we use the standard cross entropy for categorical class classification. We also use Adam as the optimizer. Finally, we can specify a metric that will be calculated when we run *evaluate()* on the model, which in this case is the accuracy. The final step is to train our model for 10 epochs (number of epochs is arbitrary).

As shown in Figure 1.1.2, the accuracy reaches very high (around 99%) just within a few epochs. This accuracy is way above the minimum accuracy set for this subtask (90%), just by a few simple layers of the CNN model.

```
In [24]: train_model.compile(optimizer='adam',
                             loss='sparse_categorical_crossentropy',
                             metrics=['accuracy'])
         train_model.fit(x=x_train, y=y_train, epochs=10)
```

```
Epoch 1/10
60000/60000 [==============================] - 44s 738us/sample - loss: 0.0319 - accuracy: 0.9896
Epoch 2/10
60000/60000 [==============================] - 41s 684us/sample - loss: 0.0260 - accuracy: 0.9913-
Epoch 3/10
60000/60000 [==============================] - 37s 610us/sample - loss: 0.0218 - accuracy: 0.9926
Epoch 4/10
60000/60000 [==============================] - 43s 722us/sample - loss: 0.0206 - accuracy: 0.9931
Epoch 5/10
60000/60000 [==============================] - 41s 684us/sample - loss: 0.0178 - accuracy: 0.9941
Epoch 6/10
60000/60000 [==============================] - 44s 739us/sample - loss: 0.0162 - accuracy: 0.9945
Epoch 7/10
60000/60000 [==============================] - 38s 630us/sample - loss: 0.0164 - accuracy: 0.9944
Epoch 8/10
60000/60000 [==============================] - 38s 642us/sample - loss: 0.0149 - accuracy: 0.9948
Epoch 9/10
60000/60000 [==============================] - 40s 661us/sample - loss: 0.0132 - accuracy: 0.9957
Epoch 10/10
60000/60000 [==============================] - 40s 662us/sample - loss: 0.0141 - accuracy: 0.9950
```

Figure 1.1.2: Model training of the CNN model.

Classification algorithm 2 – Support Vector Machine (SVM)

Support Vector Machine (SVM) is a supervised machine learning algorithm which can be used for both classification or regression problems. It performs classification by finding the hyperplane that maximizes the margin between the two classes.

To simplify the problem, as well as potentially computational efficiency, we convert images from grayscale to black and white by replacing all values in the training and testing data above 0 (i.e. not completely white) to 1 (i.e. black).

```
In [51]: # To simply the problem, converting images to black and white
         # from gray scale by replacing all values > 0 to 1.
         x_train_b = x_train
         x_test_b = x_test

         x_test_b[x_test_b>0]=1
         x_train_b[x_train_b>0]=1
```

Figure 1.1.3: Converting images to black and white from grayscale by replacing all values above 0 to 1.

Further, we convert 1D array to 2D 28x28 array using reshape, to plot and view binary images.

```
In [52]: # And Converting 1D array to 2D 28x28 array using reshape , to plot and view binary images.

for x in range(0,4):
    train_0=x_train_b[y_train==x]
    data_new=[]
    for idx in train_0.index:
        val=train_0.loc[idx].values.reshape(28,28)
        data_new.append(val)
    plt.figure(figsize=(25,25))
    for x in range(1,5):
        ax1=plt.subplot(1, 20, x)
        ax1.imshow(data_new[x],cmap='binary')
```

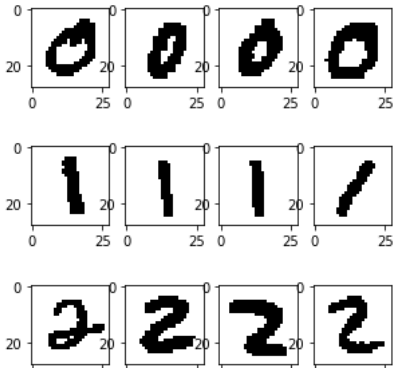

Figure 1.1.4: Converting 1D array to 2D 28x28 array using reshape, to plot and view binary images.

In order to reduce dimension, as well as potentially computational efficiency, we apply PCA in advance before implementing the SVM algorithm. We first need to standardize the data by using *StandardScaler* from the *sklearn* library. The PCA keeps 90% of information by choosing components falling within 0.90 cumulative. As shown in Figure 1.1.5, the shape after PCA is smaller than before PCA for both training and testing data.

```
In [62]: ### Classification using SVM - Binary + Dimensionality Reduction (PCA)
         #reduce dimension with PCA
         from sklearn.preprocessing import StandardScaler
         from sklearn.decomposition import PCA as sklearnPCA

         #standardize data
         sc = StandardScaler().fit(x_train_b)
         X_std_train = sc.transform(x_train_b)
         X_std_test = sc.transform(x_test_b)

         #If n_components is not set then all components are stored
         sklearn_pca = sklearnPCA().fit(X_std_train)

In [48]: #Keeping 90% of information by choosing components falling within 0.90 cumulative
         n_comp=len(cum_var_per[cum_var_per <= 0.90])
         print("Keeping 90% Info with ",n_comp," components")
         sklearn_pca = sklearnPCA(n_components=n_comp)
         x_train_pca_b = sklearn_pca.fit_transform(X_std_train)
         x_test_pca_b = sklearn_pca.transform(X_std_test)
         print("Shape before PCA for Train: ",X_std_train.shape)
         print("Shape after PCA for Train: ",x_train_pca_b.shape)
         print("Shape before PCA for Test: ",X_std_test.shape)
         print("Shape after PCA for Test: ",x_test_pca_b.shape)

Keeping 90% Info with  301  components
Shape before PCA for Train:  (60000, 784)
Shape after PCA for Train:  (60000, 301)
Shape before PCA for Test:  (10000, 784)
Shape after PCA for Test:  (10000, 301)
```

Figure 1.1.5: Applying PCA to reduce dimensions by choosing components falling within 0.90 cumulative.

The final step is to train our SVM classifier, with 42 random states, by fitting in the binary training data after PCA and the training labels. In addition, we include the time elapsed to both fit and score the model.

```
In [57]: score=[]
         fittime=[]
         scoretime=[]
         clf = svm.SVC(random_state=42)
         clf.fit(x_train_pca_b, y_train.values.ravel())

         # find the score using reduced dimensions keeping the same amount of samples, to compare accuracy.
         start_time = time.time()
         fittime = time.time() - start_time
         print("Time consumed to fit model: ",time.strftime("%H:%M:%S", time.gmtime(fittime)))
         start_time = time.time()
         score=clf.score(x_test_pca_b,y_test)
         print("Accuracy: ",score)
         scoretime = time.time() - start_time
         print("Time consumed to score model: ",time.strftime("%H:%M:%S", time.gmtime(scoretime)))
         case4=[score,fittime,scoretime]

         Time consumed to fit model:  00:00:12
         Accuracy:  0.9209
         Time consumed to score model:  00:00:14
```

Figure 1.1.6: Training SVM classifier and using it to score, as well as including its elapsed time to both train and score the model.

1.2. Describe the techniques, including data preparation, feature reduction, and training tricks in your classification algorithms. [10 marks]

Data preparation

1. Load data
   First, we import the necessary packages. Then, we load the MNIST data which has been converted to csv file in advance for ease of use (retrieved from https://pjreddie.com/projects/mnist-in-csv/). We assign the label column of the training data to *y_train* and of the testing data to *y_test*. The rest of the columns of the training and testing data are assigned to *x_train* and *x_test*.

```
In [1]: import numpy as np # linear algebra
        import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
        import matplotlib.pyplot as plt
        import tensorflow as tf
        from sklearn.preprocessing import StandardScaler

In [2]: data_train = pd.read_csv('mnist_train.csv')
        data_test = pd.read_csv('mnist_test.csv')
```

Figure 1.2.1: Loading data

2. Normalization
   We perform a grayscale normalization to reduce the effect of illumination's differences. Moreover, the CNN converge faster on [0..1] data than on [0..255].

```
In [8]: # Ensure the values to float so decimal points are still there after division
        # and normalize the pixel data
        x_train = x_train.astype('float32')
        x_test = x_test.astype('float32')
        x_train /= 255
        x_test /= 255
```

Figure 1.2.2: Normalization

3. Reshape

```
In [11]: # Reshaping the array to 4-dims before entering the Keras API
         x_train = x_train.reshape(x_train.shape[0],28,28,1)
         x_test = x_test.reshape(x_test.shape[0],28,28,1)
         input_shape = (28,28,1)
```

Figure 1.2.3: Reshaping

4. Split training and validation set

```
In [12]: # Extracting label from data
         y_train = data_train["label"]
         x_train = data_train.drop('label',axis=1)
         y_test = data_test["label"]
         x_test = data_test.drop('label',axis=1)

         print("x_train Shape: ",x_train.shape)
         print("y_train Shape: ",y_train.shape)
         print("x_label Shape: ",x_test.shape)
         print("y_label Shape: ",y_test.shape)
```

Figure 1.2.4: Splitting training and validation set

5. Try visualizing the data

```
In [39]: # check the dataset
         # printing the value of y and the image
         idx = 1 #index
         print("Value of y index",idx,"is",y_train[idx])
         plt.imshow(x_train[idx], cmap='Greys')
         plt.show()
```

```
Value of y index 1 is 0
```

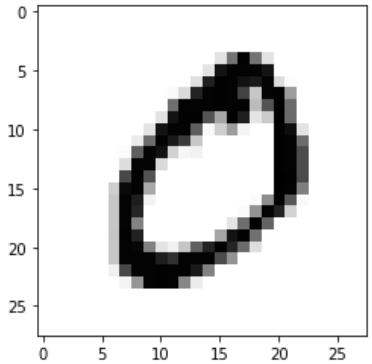

Figure 1.2.5: Data visualization example to make sure data is pre-processed correctly

Feature Reduction

1. For CNN, I used the pooling layer which reduces the dimension. At the same time, this reduces computational cost and to some extent also reduce overfitting, as mentioned before. This would especially be important if the dimensions are high.
2. Another layer I used for CNN is the dropout layer. It is a regularization method where a proportion of nodes in the layer are randomly ignored (setting their weights to zero) for each training sample. This drops randomly a proportion of the network and forces the network to learn features in a distributed way. This technique also improves generalization and reduces the overfitting.
3. As shown in Figure 1.1.3, we converted the images to black and white from grayscale by replacing all values above 0 to 1.
4. As shown in Figure 1.1.5, we applied the PCA to keeping 90% of information by choosing components falling within 0.90 cumulative.

Another training trick I used is the parameters Gamma and C for SVM (see Figure 1.2.6). Gamma is the parameter of a Gaussian Kernel (to handle non-linear classification) and C the parameter for the soft margin cost function, also known as cost of misclassification. A large C gives you low bias and high variance and vice versa.

To find optimal combination of parameters to achieve maximum accuracy, using *GridSearchCV* from *sklearn* library. *GridSearchCV* does exhaustive search over specified parameter values for an estimator. Storing values of parameters to be passed to *GridSearch* in parameters, keeping cross-validation folds as 3 and passing SVM as estimator.

Consequently, by passing the optimal parameters to classifier, we could reduce the time consumed to fit model from 8:24 minutes to 4:57 minutes. Furthermore, we have significantly increased the accuracy from 92.09% (see Figure 1.1.6) to 97.2%.

```
In [59]: clf.get_params
         parameters = {'gamma': [1, 0.1, 0.01, 0.001],
                       'C': [1000, 100, 10, 1]}
         p = GridSearchCV(clf , param_grid=parameters, cv=3)
         X=x_train_pca_b[:i]
         y=y_train[:i].values.ravel()
         start_time = time.time()
         p.fit(X,y)
         elapsed_time = time.time() - start_time
         print("Time consumed to fit model: ",time.strftime("%H:%M:%S", time.gmtime(elapsed_time)))

         Time consumed to fit model:  00:08:24

In [61]: # To verify, lets pass the optimal parameters to Classifier and check the score.
         C=p.best_params_['C']
         gamma=p.best_params_['gamma']
         clf=svm.SVC(C=C,gamma=gamma, random_state=42)

         start_time = time.time()
         clf.fit(x_train_pca_b, y_train.values.ravel())
         elapsed_time = time.time() - start_time
         print("Time consumed to fit model: ",time.strftime("%H:%M:%S", time.gmtime(elapsed_time)))
         print("Accuracy for binary: ",clf.score(x_test_pca_b,y_test))

         Time consumed to fit model:  00:04:57
         Accuracy for binary:  0.972
```

Figure 1.2.6: Applying Gamma and C parameters for SVM.

1.3. Analyse some other techniques that can be applied in your classification algorithms to improve your model's performance such as accuracy, efficiency, and storage. [10 marks]

Some of the techniques that have improved my model's accuracy, efficiency, and storage have been discussed extensively in subtask 1.2. Some general ones are converting grayscale to binary and applying PCA. A nontrivial one is applying Gamma and C parameters for SVM.

For CNN, we can add more pooling layers and dropout layers, especially if the dimensions are high. These techniques can be used to reduce computational cost (thus improving efficiency and reducing storage), reduce overfitting and improve generalization.

An excellent technique to significantly improve accuracy is data augmentation. In order to avoid overfitting problem, we can expand artificially our handwritten digit dataset. We can make your existing dataset even larger. The idea is to alter the training data with small transformations to reproduce the variations occurring when someone is writing a digit. For example, the number is not centered, the scale is not the same (some who write with big/small numbers), the image is rotated. Approaches that alter the training data in ways that change the array representation while keeping the label the same are known as data augmentation techniques. Some popular augmentations people use are grayscales, horizontal flips, vertical flips, random crops, color jitters, translations, rotations, and much more. By applying just a couple of these transformations to our training data, we can easily double or triple the number of training examples and create a very robust model.

Another general technique to reduce computational cost is feature scaling. Before making any actual predictions, it is always a good practice to scale the features so that all of them can be uniformly evaluated. Wikipedia explains the reasoning pretty well: "Since the range of values of raw data varies widely, in some machine learning algorithms, objective functions will not work properly without normalization. For example, the majority of classifiers calculate the distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature. Therefore, the range of all features should be normalized so that each feature contributes approximately proportionately to the final distance." The gradient descent algorithm (which is used in neural network training and other machine learning algorithms) also converges faster with normalized features. Thus, this would increase the computational efficiency.

Task 2: Support Vector Machine (SVM) and Principal Component Analysis (PCA) [40 Marks]

In this task, we are going to classify iris species by classifying the iris dataset with support vector machine (SVM) and further analyze it with different combinations of principal components of PCA.

The dataset consists of four attributes: sepal-width, sepal-length, petal-width and petal-length. These are the attributes of specific types of iris plant. The task is to predict the class to which these plants belong. There are three classes in the dataset: Iris-setosa, Iris-versicolor and Iris-virginica.

2.1. Using the iris.data, select the training dataset and validation dataset, and implement the SVM algorithm (based on public packages or libraries) to classify the types of iris (achieving an accuracy of 90%). [10 marks]

As shown in Figure 2.1.1, we start by importing the necessary packages, especially from the *sklearn* library that is particularly useful for SVM. Then, we import the iris dataset that is available from *sklearn* datasets. The next step is to pre-process the data by assigning the data features *iris.data* to a variable *x* and the data targets *iris.target* to a variable *y*.

To avoid overfitting, we will divide our dataset into training and test splits, which gives us a better idea as to how our algorithm would perform during the testing phase. This way, our algorithm is tested on unseen data, as it would be in a production application. Thus, we split the dataset into 80% training data and 20% testing data. This means that out of 150 records, the training set will contain 120 of those records and the test set contains 30 records. Further, we also split x and y into training and testing data respectively.

Before making any actual predictions, it is always a good practice to scale the features so that all of them can be uniformly evaluated. Thus we use *StandardScaler* from *sklearn.preprocesing* to standardize the features of the training data *x_train* and of the test data *x_test* into new variables *x_std_train* and *x_std_test* respectively.

After preprocessing the data, now we train the classifier using *SVC* from *sklearn.svm* by fitting in the standardized training set *x_std_train* and the training label set *y_train*.

```
In [478]: # Importing necessary libraries
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler

# Loading the iris dataset
iris = datasets.load_iris()

# x -> features, y -> label
x = iris.data
y = iris.target

# Dividing x, y into train and test data
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.20)

# Feature scaling: standardize data
sc = StandardScaler().fit(x_train)
x_std_train = sc.transform(x_train)
x_std_test = sc.transform(x_test)

# Training SVM classifier
modelSVC = SVC()
modelSVC.fit(x_std_train, y_train)
```

Figure 2.1.1: Importing libraries, importing the dataset, data pre-processing, feature scaling, training SVM classifier.

After training the SVM classifier, we then use the SVM classifier to predict the standardized testing data which returns labels that are stored in a variable *y_pred*. The next step is to evaluate the SVM classifier performance, i.e. its confusion matrix and classification report.

As shown in Figure 2.1.2, the SVM classifier yields an excellent overall accuracy of 97%. In particular, the confusion matrix shows that the model:

- correctly identified all 0 classes as 0's
- correctly identified all 1 classes as 1's
- correctly classified 12 class 2's but miss-classified 1 class 2's as class 1

```
In [479]: y_pred = modelSVC.predict(x_std_test)
          from sklearn.metrics import classification_report, confusion_matrix
          print(confusion_matrix(y_test, y_pred))
          print(classification_report(y_test, y_pred))

          [[ 6  0  0]
           [ 0 11  0]
           [ 0  1 12]]
                        precision    recall  f1-score   support

                     0       1.00      1.00      1.00         6
                     1       0.92      1.00      0.96        11
                     2       1.00      0.92      0.96        13

              accuracy                           0.97        30
             macro avg       0.97      0.97      0.97        30
          weighted avg       0.97      0.97      0.97        30
```

Figure 2.1.2: Performance metrics of the SVM classifier.

2.2. Using the same iris.data, reduce the dimension of features applying the PCA and extract the first, second, and third principal components. [10 marks]

Using the standardized training *x_std_train* and testing data *x_std_test* from the subtask 2.1, we extract 3 principal components and store them into new variables *x_train_pca* and *x_test_pca* respectively, as shown in Figure 2.2.1. We then train the SVM classifier again with the new training data with the three principal components and the training labels.

```
In [484]: from sklearn.decomposition import PCA

          ### reduce dimension with PCA -----------------------
          #choose 3 components
          sklearn_pca = PCA(n_components=3)
          x_train_pca = sklearn_pca.fit_transform(x_std_train)
          x_test_pca = sklearn_pca.transform(x_std_test)
          ### -------------------------------------------------

          # Training SVM classifier with PCA
          modelSVMPCA = SVC()
          modelSVMPCA.fit(x_train_pca, y_train)
```

Figure 2.2.1: SVM classifier with PCA.

After training the SVM classifier with PCA, we then use the SVM classifier with PCA to predict the standardized testing data also with the three principal components which returns labels that are stored in a new variable *y_pred_pca*. The next step is to evaluate the SVM classifier with PCA performance, i.e. its confusion matrix and classification report.

As shown in Figure 2.2.2, the SVM classifier with PCA yields the same overall accuracy of 97% as the SVM classifier without PCA in task 2.1. In particular, the confusion matrix is also the same as in task 2.1. This means applying PCA to the SVM classifier and reducing its original 4 features or dimensions to three principal components does not (significantly, if any) affect the accuracy.

```
In [483]: y_pred_pca = modelSVMPCA.predict(x_test_pca)
          print(confusion_matrix(y_test, y_pred_pca))
          print(classification_report(y_test, y_pred_pca))
```

```
[[ 6  0  0]
 [ 0 11  0]
 [ 0  1 12]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         6
           1       0.92      1.00      0.96        11
           2       1.00      0.92      0.96        13

    accuracy                           0.97        30
   macro avg       0.97      0.97      0.97        30
weighted avg       0.97      0.97      0.97        30
```

Figure 2.2.2: Performance metrics of the SVM classifier with three principal components.

To verify the output of the model with the three principal components, we visualize it as shown in Figure 2.2.3. We plot a 3-dimensional graph in which the first principal component takes the x-axis, the second principal component takes the y-axis, and the third principal component takes the z-axis. It seems quite apparent that the training data can be classified through a plane that cuts through the first principal component alone. Another important observation is that the first principal component indeed seems to have the highest variance.

```
In [14]: import matplotlib.pyplot as plt
         from mpl_toolkits.mplot3d import Axes3D

         fig = plt.figure(1, figsize=(8, 6))
         ax = Axes3D(fig, elev=-150, azim=110)
         ax.scatter(x_train_pca[:, 0], x_train_pca[:, 1], x_train_pca[:, 2], c=y_train,
                    cmap=plt.cm.coolwarm, edgecolor='k', s=40)
         ax.set_title("First three PCA directions for training data")
         ax.set_xlabel("Principal component 1")
         ax.w_xaxis.set_ticklabels([])
         ax.set_ylabel("Principal component 2")
         ax.w_yaxis.set_ticklabels([])
         ax.set_zlabel("Principal component 3")
         ax.w_zaxis.set_ticklabels([])
         ax.show()
```

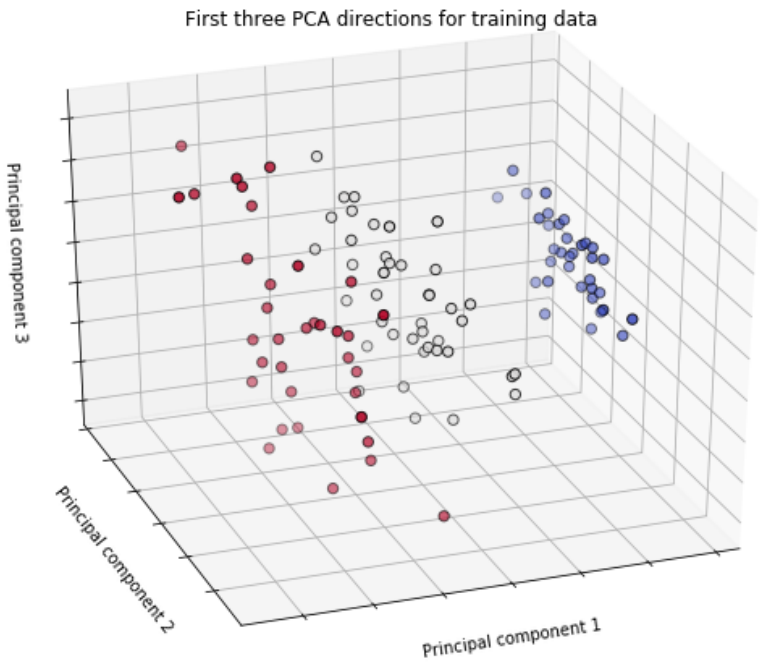

Figure 2.2.3: A 3D representation of the training data with the three principal components as the three axes.

The variable *x_train_pca* contains an array of the three principal components of the training data (see Figure 2.2.4). Thus, to extract the first, second, and third principal component, we just extract the first, second, and third column of the array in *x_train_pca* as shown in Figure 2.2.5. The same case goes for the testing data *x_test_pca*.

```
In [491]: x_train_pca

Out[491]: array([[ 1.47169466,  1.11150676,  0.8753308 ],
                 [-2.13874679,  0.46205556, -0.12022836],
                 [-2.0687003 , -0.74062156, -0.20635113],
                 [-1.6989035 ,  0.42747991, -0.28790358],
                 [ 1.2968531 , -0.59693671,  0.50135434],
                 [ 0.75459196, -1.66028395, -0.78869381],
                 [-2.11071772,  0.98040323, -0.17995133],
                 [-2.32465659,  0.01556118,  0.36237566],
                 [-2.2009426 ,  0.10782167,  0.12151702],
                 [ 0.22917583, -0.11670073,  0.12244128],
                 [ 1.24881383,  0.97991585, -0.75427218],
                 [ 1.75309543, -0.29405655, -0.03115702],
                 [-1.92650752,  0.22990718,  0.09243669],
                 [-2.03844218,  0.25553543, -0.16645241],
                 [ 2.90261855,  0.97598963, -0.6715637 ],
                 [ 0.3197239 , -0.71622232, -0.3076906 ],
                 [-2.27134464,  0.62057986,  0.02922342],
                 [-2.6630005 ,  0.41369133,  0.35527933],
                 [ 1.21071846, -0.5307701 ,  0.08312057],
```

Figure 2.2.4: x_train_pca is an array containing the three principal components in the training data, each principal component in each column.

```
In [492]: ## PC 1
          PC1_train = x_train_pca[:,[0]]
          PC1_test = x_test_pca[:,[0]]

          ## PC 2
          PC2_train = x_train_pca[:,[1]]
          PC2_test = x_test_pca[:,[1]]

          ## PC 3
          PC3_train = x_train_pca[:,[2]]
          PC3_test = x_test_pca[:,[2]]
```

Figure 2.2.5: Extracting each principal component from the column of x_train_pca and x_test_pca.

To verify this, as shown in Figure 2.2.6, the variable *PC1_train* that contains the first principal component, is the first column of the *x_train_pca* array in Figure 2.2.4.

```
In [510]: PC1_train

Out[510]: array([[ 1.47169466],
                 [-2.13874679],
                 [-2.0687003 ],
                 [-1.6989035 ],
                 [ 1.2968531 ],
                 [ 0.75459196],
                 [-2.11071772],
                 [-2.32465659],
                 [-2.2009426 ],
                 [ 0.22917583],
                 [ 1.24881383],
                 [ 1.75309543],
                 [-1.92650752],
                 [-2.03844218],
                 [ 2.90261855],
                 [ 0.3197239 ],
                 [-2.27134464],
                 [-2.6630005 ],
                 [ 1.21071846],
```

Figure 2.2.5: PC1_train that contains the first principal component, is the first column of the x_train_pca array as shown in Figure 2.2.3.

2.3. Using the extracted first, second, and third principal component in Task 2.2, respectively, train an SVM model to classify the types of iris and compare their accuracies. [10 marks]

In this subtask, we are comparing the accuracies of the SVM classifier using only any one of the three principal components extracted by PCA. First, as shown in Figure 2.3.1, the weighted accuracy of the SVM classifier with the first principal component is 90%, which is still very high especially considering that the classifier only predicts with one parameter (i.e. principal component). It is also the highest accuracy amongst other individual principal component as shown in Figures 2.3.2 and 2.3.3. The first principal component was indeed expected to have the highest accuracy as it is a compressed combination of original predictor variables which captures the maximum variance in the data set.

```
In [497]: modelSVMPC1 = SVC()
          modelSVMPC1.fit(PC1_train, y_train)
          y_pred_pc1 = modelSVMPC1.predict(PC1_test)
          print(confusion_matrix(y_test, y_pred_pc1))
          print(classification_report(y_test, y_pred_pc1))
```

```
[[ 6  0  0]
 [ 0  9  2]
 [ 0  1 12]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         6
           1       0.90      0.82      0.86        11
           2       0.86      0.92      0.89        13

    accuracy                           0.90        30
   macro avg       0.92      0.91      0.92        30
weighted avg       0.90      0.90      0.90        30
```

Figure 2.3.1: Performance metrics of the SVM classifier with the first principal component.

Second, as shown in Figure 2.3.2, the weighted accuracy of the SVM classifier with the second principal component is 22%, which is the lowest accuracy. It misclassifies many of all the classes.

```
In [498]: modelSVMPC2 = SVC()
          modelSVMPC2.fit(PC2_train, y_train)
          y_pred_pc2 = modelSVMPC2.predict(PC2_test)
          print(confusion_matrix(y_test, y_pred_pc2))
          print(classification_report(y_test, y_pred_pc2))
```

```
[[ 4  2  0]
 [ 8  3  0]
 [12  1  0]]
              precision    recall  f1-score   support

           0       0.17      0.67      0.27         6
           1       0.50      0.27      0.35        11
           2       0.00      0.00      0.00        13

    accuracy                           0.23        30
   macro avg       0.22      0.31      0.21        30
weighted avg       0.22      0.23      0.18        30
```

Figure 2.3.2: Performance metrics of the SVM classifier with the second principal component.

Third, as shown in Figure 2.3.3, the weighted accuracy of the SVM classifier with the third principal component is 68%, which is moderate especially considering that it only uses one parameter (i.e. principal component) to predict. It correctly classifies all the 0 classes as 0's, although misclassifies some of the other classes.

```
In [504]: modelSVMPC3 = SVC()
          modelSVMPC3.fit(PC3_train, y_train)
          y_pred_pc3 = modelSVMPC3.predict(PC3_test)
          print(confusion_matrix(y_test, y_pred_pc3))
          print(classification_report(y_test, y_pred_pc3))
```

```
[[6 0 0]
 [7 4 0]
 [7 4 2]]
              precision    recall  f1-score   support

           0       0.30      1.00      0.46         6
           1       0.50      0.36      0.42        11
           2       1.00      0.15      0.27        13

    accuracy                           0.40        30
   macro avg       0.60      0.51      0.38        30
weighted avg       0.68      0.40      0.36        30
```

Figure 2.3.3: Performance metrics of the SVM classifier with the third principal component.

2.4. For each combination of the extracted first, second, and third principal components, train a SVM model to classify the types of iris, and then compare their accuracies. [10 marks]

In this subtask, we are comparing the accuracies of the SVM classifier using only any two of the three principal components extracted by PCA. First, as shown in Figure 2.4.1, the weighted accuracy of the SVM classifier with the first and second principal components is 90%, which is not (significantly, if any) different from the accuracy of the SVM classifier with only the first principal component (see Figure 2.3.1). This may make sense as the SVM classifier with only the second principal component yields a very low accuracy of 22% and thus does not significantly contribute to the combination.

```
In [512]: ## PC 1 & 2
          PC12_train = x_train_pca[:,[0,1]]
          PC12_test = x_test_pca[:,[0,1]]
          modelSVMPC12 = SVC()
          modelSVMPC12.fit(PC12_train, y_train)
          y_pred_pc12 = modelSVMPC12.predict(PC12_test)
          print(confusion_matrix(y_test, y_pred_pc12))
          print(classification_report(y_test, y_pred_pc12))
```

```
[[ 6  0  0]
 [ 0  9  2]
 [ 0  1 12]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         6
           1       0.90      0.82      0.86        11
           2       0.86      0.92      0.89        13

    accuracy                           0.90        30
   macro avg       0.92      0.91      0.92        30
weighted avg       0.90      0.90      0.90        30
```

Figure 2.4.1: Performance metrics of the SVM classifier with the first and second principal components.

To verify this, we plot the training data classified with decision boundary by the SVM classifier with principal components 1 and 2, as shown in Figure 2.4.2. It is quite apparent that the model correctly

classifies many instances but misclassifies some of the red and white instances. Note that the code in Figure 2.4.2 can be reused for the visualizing other two combinations of the principal components as shown in Figures 2.4.4 and 2.4.6.

In [21]:
```
import numpy as np

# Visualizing the modeled svm classifiers with Iris Sepal features
h = .02  # step size in the mesh

# create a mesh to plot in
x_min, x_max = x_train_pca[:, 0].min() - 1, x_train_pca[:, 0].max() + 1
y_min, y_max = x_train_pca[:, 1].min() - 1, x_train_pca[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# Plot the decision boundary. For that, we will assign a color to each point in the mesh [x_min, x_max]*[y_min, y_max]
Z = modelSVMPC12.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)

# Plot also the training points
plt.scatter(x_train_pca[:, 0], x_train_pca[:, 1], c=y_train, cmap=plt.cm.coolwarm)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('SVM and PCA with Principal Components 1 and 2')

plt.show()
```

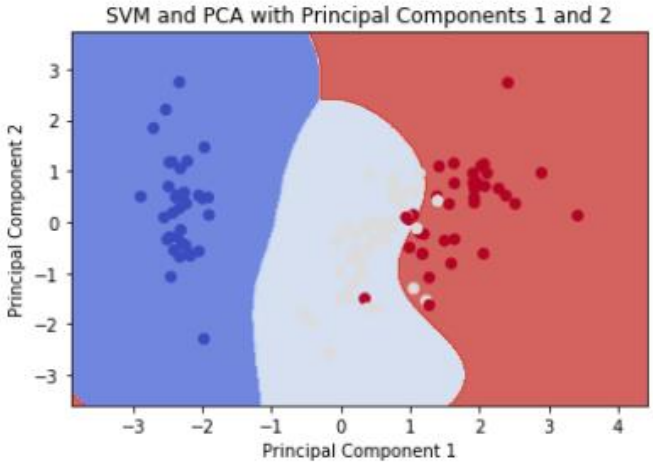

Figure 2.4.2: Training data classified with decision boundary by the SVM classifier with principal components 1 and 2.

Second, as shown in Figure 2.4.3, the weighted accuracy of the SVM classifier with the first and third principal components is 97%, which is the highest accuracy of the SVM classifier with any of the two principal components. This makes sense since the SVM classifier with only the third principal component has the second highest accuracy of 68% (see Figure 2.3.3) after the first principal component. In combination, both the first and third principal components yield an even higher accuracy. In fact, the accuracy is similar (or the same in this case) as the SVM accuracy without PCA (see Figure 2.1.2) and with all the three principal components (see Figure 2.2.2). Based on this information, we may then infer that using only the first and third principal components is sufficient to classify the iris dataset with similar accuracy.

```
In [513]: ## PC 1 & 3
PC13_train = x_train_pca[:,[0,2]]
PC13_test = x_test_pca[:,[0,2]]
modelSVMPC13 = SVC()
modelSVMPC13.fit(PC13_train, y_train)
y_pred_pc13 = modelSVMPC13.predict(PC13_test)
print(confusion_matrix(y_test, y_pred_pc13))
print(classification_report(y_test, y_pred_pc13))
```

```
[[ 6  0  0]
 [ 0 11  0]
 [ 0  1 12]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         6
           1       0.92      1.00      0.96        11
           2       1.00      0.92      0.96        13

    accuracy                           0.97        30
   macro avg       0.97      0.97      0.97        30
weighted avg       0.97      0.97      0.97        30
```

Figure 2.4.3: Performance metrics of the SVM classifier with the first and third principal components.

To verify this, we plot the training data classified with decision boundary by the SVM classifier with principal components 1 and 3, as shown in Figure 2.4.4. It is quite apparent that the model correctly classifies most of the instances but misclassifies some of the red and white instances. This is consistent with the result we got where the combination of principal components 1 and 3 yield the highest accuracy of almost perfect 97%.

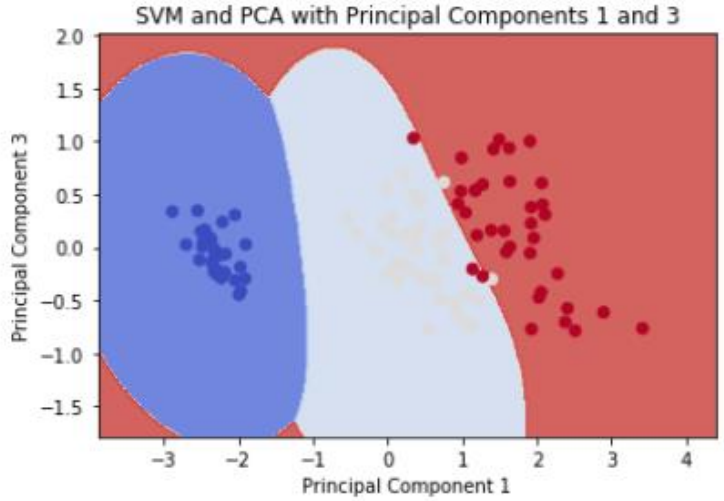

Figure 2.4.4: Training data classified with decision boundary by the SVM classifier with principal components 1 and 3.

Third, as shown in Figure 2.4.5, the weighted accuracy of the SVM classifier with the first and third principal components is 69%, which is only slightly higher than the third principal component alone (see Figure 2.3.3). This may make sense as the SVM classifier with only the second principal

component yields a very low accuracy of 22% and thus does not significantly contribute to the combination.

```
In [514]: ## PC 2 & 3
PC23_train = x_train_pca[:,[1,2]]
PC23_test = x_test_pca[:,[1,2]]
modelSVMPC23 = SVC()
modelSVMPC23.fit(PC23_train, y_train)
y_pred_pc23 = modelSVMPC23.predict(PC23_test)
print(confusion_matrix(y_test, y_pred_pc23))
print(classification_report(y_test, y_pred_pc23))
```

```
[[ 4  2  0]
 [ 8  3  0]
 [11  0  2]]
              precision    recall  f1-score   support

           0       0.17      0.67      0.28         6
           1       0.60      0.27      0.37        11
           2       1.00      0.15      0.27        13

    accuracy                           0.30        30
   macro avg       0.59      0.36      0.31        30
weighted avg       0.69      0.30      0.31        30
```

Figure 2.4.5: Performance metrics of the SVM classifier with the second and third principal components.

To verify this, we plot the training data classified with decision boundary by the SVM classifier with principal components 2 and 3, as shown in Figure 2.4.6. It is quite apparent that the model misclassifies many instances, be it blue, white, or red. This is also consistent with the result we got where the combination of principal components 2 and 3 yield the lowest accuracy of 69%.

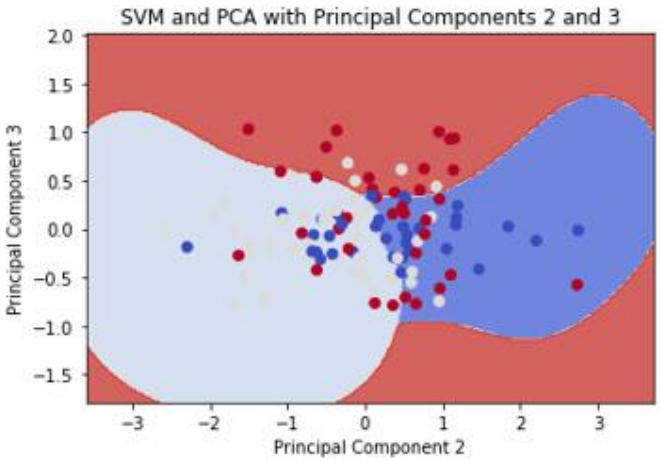

Figure 2.4.6: Training data classified with decision boundary by the SVM classifier with principal components 2 and 3.

Task 3: K-Means Clustering [25 Marks]

In this task, we are going to use the iris dataset again for our K-means clustering algorithm, and then later reducing its dimensions with three principal components of PCA.

3.1. Specify the number of clustering (e.g., 1, 2, 3, 4, 5, and 6) and implement the k-means algorithm (based on public packages or libraries) to classify the iris.data. [10 marks]

As shown in Figure 3.1.1, we start by importing the necessary packages, especially from the *sklearn* library that is particularly useful for K-means clustering. Then, we import the iris dataset that is available from *sklearn* datasets. The next step is to pre-process the data by assigning the data features *iris.data* to a variable *x* and the data targets *iris.target* to a variable *y*, after assigning columns into the data and converting it to pandas dataframe for the ease of plotting later on in Figure 3.1.6. Furthermore, we perform feature scaling so that all of our features can be uniformly evaluated, before making any actual predictions.

```
In [25]: # Import the packages
import pandas as pd
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import sklearn.metrics as sm

# Load the iris dataset that is available from sklearn
iris = datasets.load_iris()

# x -> features, y -> label
x = pd.DataFrame(iris.data, columns=['Sepal Length', 'Sepal Width', 'Petal Length', 'Petal Width'])
y = pd.DataFrame(iris.target, columns=['Target'])

# Feature scaling: standardize data
sc = StandardScaler().fit(x)
x_std = sc.transform(x)
```

Figure 3.1.1: Importing libraries, importing the dataset, and data pre-processing.

An optional step is to check the array of *iris.target*, to compare with the predicted target labels later on in the next step in Figure 3.1.3. As shown in Figure 3.1.2, *iris.target* is an array of integers used to represent the Iris species. 0=Setosa, 1=Versicolor, 2=Virginica.

```
In [116]: iris.target

Out[116]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
                 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
                 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Figure 3.1.2: Showing the array of the iris targets.

After the data has been pre-processed, we now model a K-means clustering with 3 clusters. The number of clusters is chosen to be 3, so that later we can directly compare with the actual targets *y* that is available in the original iris dataset. Next, we fit the iris data *x* into the K-means model and then show the predicted labels from the K-means model. As shown in Figure 3.1.3, the predicted labels look very similar to the *iris.target* array that is shown in figure 3.1.2.

It is important to note that the K-means model also assigns integer ids for the three clusters: 0, 1, 2. Since the K-means is an unsupervised learning algorithm, it has no knowledge of the *iris.target* data and thus the clusters being given ids [0, 1, 2] as the *iris.target* is just a coincidence.

```
In [50]: # Create a K-means model with 3 clusters
         modelKMeans = KMeans(n_clusters=3)
         modelKMeans.fit(x)
         modelKMeans.labels_

Out[50]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2,
                2, 2, 2, 1, 1, 2, 2, 2, 2, 1, 2, 1, 2, 1, 2, 2, 1, 1, 2, 2, 2, 2,
                2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 1])
```

Figure 3.1.3: Creating a K-means model with 3 clusters and showing its predicted labels of the given x iris data.

Furthermore, note that it is also a coincidence that the labels order in Figure 3.1.3 (mostly 0 first, mostly 1, and then mostly 2). The labels order generated from the model could be different, as shown in Figure 3.1.4, as the order goes from mostly 1 first, mostly 0, and then mostly 2. The reason for this is that, as mentioned previously, K-means is unsupervised so that it has no knowledge of the *iris.target* data and thus also its ordering. For the case in Figure 3.1.4, this can be manually fixed by *np.choose(model.labels_, [1, 0, 2]).astype(np.int64)* so that 0 is changed to 1 and 1 is changed to 0. However, for simplicity in the next steps, I use the label order that corresponds to the one in *iris.target* as in Figure 3.1.3.

```
In [117]: # Create a K-means model with 3 clusters
          modelKMeans = KMeans(n_clusters=3)
          modelKMeans.fit(x)
          modelKMeans.labels_

Out[117]: array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, 1, 1, 1, 1, 1, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2, 2, 2, 2, 0, 2, 2, 2,
                 2, 2, 2, 0, 0, 2, 2, 2, 2, 0, 2, 0, 2, 0, 2, 2, 0, 0, 2, 2, 2, 2,
                 2, 0, 2, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 0])
```

Figure 3.1.4: Another possibility of cluster ids assignment of the K-means model, if the code is re-run.

Given the predicted labels as shown in Figure 3.1.3, we compare the original targets *y* with the predicted labels that have been stored in a variable *y_pred*. This is to give the performance metrics

for the K-means model, as shown in Figure 3.1.5. The performance metrics contains of the confusion matrix and the classification report.

The model yields an overall accuracy of 91%, which is quite good for an unsupervised learning model. In particular, the confusion matrix shows that the model:

- correctly identified all 0 classes as 0's
- correctly classified 48 class 1's but miss-classified 2 class 1's as class 2
- correctly classified 36 class 2's but miss-classified 14 class 2's as class 1

```
In [129]: # Performance Metrics
          y_pred = modelKMeans.labels_
          print("Confusion Matrix: \n", sm.confusion_matrix(y, y_pred))
          print("Classification Report: \n", sm.classification_report(y, y_pred))
```

```
Confusion Matrix:
 [[50  0  0]
 [ 0 48  2]
 [ 0 14 36]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       0.77      0.96      0.86        50
           2       0.95      0.72      0.82        50

    accuracy                           0.89       150
   macro avg       0.91      0.89      0.89       150
weighted avg       0.91      0.89      0.89       150
```

Figure 3.1.5: Performance metrics of the K-means model.

To verify the output of the model, lets visualize it. We plot the actual classes against the predicted classes from the K Means model. As an example, we are plotting the Petal Length and Width in Figure 3.16. The two plots show a close similarity and that the results correspond to the confusion matrix in Figure 3.1.5.

```
In [113]: # Import extra packages
          import numpy as np
          import matplotlib.pyplot as plt

          #Start with a plot figure of size 12 units wide & 3 units tall
          plt.figure(figsize=(12,3))

          # Create an array of three colours, one for each species.
          colormap = np.array(['red', 'green', 'blue'])

          # The fudge to reorder the cluster ids.
          predictedY = np.choose(model.labels_, [1, 0, 2]).astype(np.int64)

          # Plot the classifications that we saw earlier between Petal Length and Petal Width
          plt.subplot(1, 2, 1)
          plt.scatter(x['Petal Length'], x['Petal Width'], c=colormap[y['Target']], s=40)
          plt.title('Before classification')

          # Plot the classifications according to the model
          plt.subplot(1, 2, 2)
          plt.scatter(x['Petal Length'], x['Petal Width'], c=colormap[y_pred], s=40)
          plt.title("Model's classification")
```

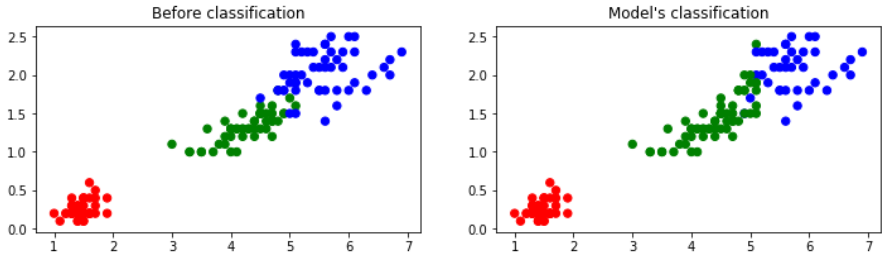

Figure 3.1.6: Plot comparison of the original classification and the K-means model classification.

3.2. Apply the PCA to reduce the dimension of features and combine the first, second, and third principal components to implement the k-means algorithm (based on public packages or libraries) to classify the iris.data. [15 marks]

In this subtask, we use the same dataset as subtask 3.1. To reduce the feature dimensions by applying PCA, we first import the necessary packages from *sklearn*. Note that we have performed feature scaling in advance as *x_std* (see Figure 3.1.1), so we do not have to standardize the data again. The next step of PCA is to the number of principal components we want to extract and store them into a new variable *x_pca*. After obtaining the principal components, we create a K-means model with 3 clusters also and then fit the data with principal components *x_pca* into the model.

```
In [110]: # Importing necessary libraries for PCA
          from sklearn.preprocessing import StandardScaler
          from sklearn.decomposition import PCA

          # Choose 3 principal components
          sklearn_pca = PCA(n_components=3)
          x_pca = sklearn_pca.fit_transform(x_std)

          # Create a K-means model with 3 clusters
          modelKMeansPCA = KMeans(n_clusters=3)
          modelKMeansPCA.fit(x_pca)
```

Figure 3.2.1: K-means of 3 clusters with PCA.

After modelling K-means with PCA, we evaluate its performance by computing its performance metrics (by confusion matrix and classification report), as shown in Figure 3.2.2. We compare the original targets *y* with the predicted labels from the K-means model with PCA that have been stored in a variable *y_pred_PCA*.

The model yields an overall accuracy of 83%, which is lower than the K-means model without PCA, but still quite good for an unsupervised learning model especially with reduced dimensions (from 4 to 3). This drop in accuracy is because the model misclassified 9 more of class 1's, whereas the model correctly classified the same number as the K-means model without PCA. In particular, the confusion matrix shows that the model:

- correctly identified all 0 classes as 0's
- correctly classified 39 class 1's but miss-classified 11 class 1's as class 2
- correctly classified 36 class 2's but miss-classified 14 class 2's as class 1

```
In [111]: # Performance Metrics
          y_pred_pca = modelKMeansPCA.labels_
          print(confusion_matrix(y, y_pred_pca))
          print(classification_report(y, y_pred_pca))
```

```
[[50  0  0]
 [ 0 39 11]
 [ 0 14 36]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       0.74      0.78      0.76        50
           2       0.77      0.72      0.74        50

    accuracy                           0.83       150
   macro avg       0.83      0.83      0.83       150
weighted avg       0.83      0.83      0.83       150
```

Figure 3.2.2: Performance metrics of the K-means model with PCA.