# Iris Flower Classification Using Principal Component Analysis and Supervised Learning Methods

# Project

**Computational Intelligence 1**

**Student**
Bianca-Alexandra GHIORGHIU

**Coordinator**
Victor Emil NEAGOE


January 18, 2023

# Contents

# Introduction

## Project Description

The purpose of this project is to use two supervised learning approaches for classification on the Iris Flower Dataset: k-Nearest Neighbours and Nearest Prototype. Furthermore, Principal Component Analysis will be performed to reduce dimensionality.

The effects of applying PCA will be observed and the two classification algorithms will be compared, taking into consideration their complexity, efficiency and accuracy.

## Structure

- **Chapter 1** provides a short introduction about machine learning and the necessary theoretical context for understanding the k-Nearest Neighbours, Nearest Prototype and Principal Component Analysis algorithms.
- **Chapter 2** outlines the steps taken in order to implement the project, from the data preprocessing and exploration part to the implementation of the principal component analysis algorithm for dimensionality reduction and the k-Nearest Neighbors and Nearest Prototype (Nearest Centroid) classification algorithms.
- The optimization process was outlined in **Chapter 3**. A grid search was used to determine the best way to split the data and the best distance metric for each of the three algorithms in the following cases: no PCA and PCA with $n = 1, \ldots, 4$ components.
- The obtained results are analyzed in **Chapter 4**. First, we investigate the accuracy and confusion matrix of the two algorithms without using PCA. Then, because we can only have a maximum of four principal components (because the Iris dataset only has four features), we examine the results obtained by keeping $n = 1, \ldots, 4$ principal components. Taking into consideration the obtained results, the three implemented algorithms are compared and the effect of applying PCA is analysed.
- The main use cases of principal component analysis are outlined in the **Conclusion** chapter, along with a comparison of the k-NN and Nearest Centroid algorithms.

# Chapter 1

# Theoretical Background

## 1.1 Machine Learning (ML)

Machine learning is a field of artificial intelligence that focuses on the study of methods that enable systems to improve themselves autonomously from experience [1]. There are the three fundamental components of machine learning:

**Dataset:** Data samples being used train machine learning algorithms.

**Features:** Essential bits of information, as they indicate to the system what to focus on.

**Algorithm:** The same task can be addressed with distinct algorithms. The criteria for precision, quality of results, and processing power depend on the algorithm chosen. It is also worth noting that combining two or more algorithms can occasionally improve performance (ensemble learning).

The five principal categories of machine learning are supervised learning, unsupervised learning, semi-supervised learning, reinforcement learning, and dimensionality reduction.

### 1.1.1 Supervised Learning

After analyzing large amounts of data with correct labels and identifying the relationships between data that yield the correct answers, supervised learning systems can make predictions. The systems are referred to as "supervised" since they are given data with proper labels. They make predictions, which are corrected by the label. Regression and classification are the two most prominent applications of supervised learning [2].

- **Regression:** One attempts to predict a continuous outcome by mapping input variables to a continuous function.
- **Classification:** In a classification task, one attempts to predict discrete outcomes. Essentially, input variables are mapped to discrete classes. There are two types of classification: binary-classification, which deals with two classes, and multiclass-classification, which handles several classes.

### 1.1.2 Unsupervised Learning

Unsupervised learning enables one to approach issues with little or no knowledge of the expected outcomes. By clustering the data based on the correlations between the variables, one may generate structure from data in which the effects of the variables are unclear. Clustering differs from classification in that the user does not specify the categories. In unsupervised learning, there is also no feedback based on the accuracy of the predictions [2].

### 1.1.3 Semi-supervised Learning

Semi-supervised learning lies between unsupervised and supervised learning, with the primary characteristic being that some samples lack labels. Numerous machine-learning researchers have shown that using unlabeled data together with a modest bit of labeled data may significantly enhance learning accuracy.

### 1.1.4 Reinforcement Learning

Models based on reinforcement learning generate predictions by receiving rewards or penalties for actions taken within a domain. A reinforcement learning model creates a policy that specifies the optimal approach for maximizing rewards [3].

### 1.1.5 Dimensionality Reduction

Dimensionality reduction is the process of lowering the number of considered features by decomposing them into a set of principal values. Principal component analysis (PCA) is one of the most widely used techniques for dimensionality reduction; it entails transforming high-dimensional data into a smaller space [4].

## 1.2 Principal Component Analysis

The process of linear or nonlinear transformation of the original space of observations into a low-dimensional space in which the data classification algorithm runs is defined as feature selection. Some classification, exploration or analysis methods that are efficient in a low-dimensional environment may become impractical in a high-dimensional space, requiring this transition. Furthermore, because real-time decisions are often needed, minimizing computational volume is critical [5].

Principal Component Analysis (PCA), commonly known as the Karhunen-Loève Transform, is an effective statistical approach for feature selection that has become a standard in data mining applications in recent decades [5]. It is a technique for reducing dataset dimensionality, enhancing interpretability, and minimizing information loss. This is achieved by linearly transforming the data into a new coordinate system in which (most of) the variance in the data can be represented with fewer dimensions than the original data [6].

# Algorithm

Knowing that we have the n-dimensional set of vectors $X_1, \ldots, X_L$, the PCA algorithm steps are as following [5]:

**Step 1:** Calculate the covariance matrix for the features in the dataset

$$\Sigma_X = \frac{1}{L} \sum_{i=1}^{L} (X_i - \mu_x)(X_i - \mu_x)^t \tag{1.1}$$

where

$$\mu_x = \frac{1}{L} \sum_{i=1}^{L} X_i \tag{1.2}$$

**Step 2:** Calculate the eigenvalues for the covariance matrix

The eigenvalues $\lambda_i$, $i = 1, \ldots, n$ of the matrix $\Sigma_X$, are determined as as the solution of the equation:

$$|\Sigma_X - \lambda \cdot I_n| = 0 \tag{1.3}$$

where $I_n$ is the n-dimensional identity matrix.

The roots of equation 1.3, representing the eigenvalues, are ordered in descending order the largest m eigenvalues are retained.

**Step 3:** Calculate the eigenvectors for the covariance matrix

The eigenvectors are determined by solving the following system of equations:

$$\begin{cases} \Sigma_X \cdot \Phi_i = \lambda_i \cdot \Phi_i \\ \Phi_i^t \cdot \Phi_i = 1 \end{cases}$$

where $\Phi_1, \ldots, \Phi_m, \Phi_{m+1}, \ldots, \Phi_n$ are the eigenvectors of $\Sigma_X$, corresponding to eigenvalues in descending order

**Step 4:** The eigenvector matrix K is constructed

$$K = (\Phi_1 | \ldots | \Phi_n)^t \tag{1.4}$$

**Step 5:** The PCA matrix is constructed

$$PCA = (\Phi_1 | \ldots | \Phi_m)^t \tag{1.5}$$

where m is the number of features wanted to be kept after applying PCA.

**Step 6:** Transform the original matrix

Applying the PCA transformation yields vectors with reduced dimensionality in m-dimensional space (where $m < n$).

$$Z_i = PCA \cdot X_i \tag{1.6}$$

## 1.3 k-Nearest Neighbors

The k-Nearest Neighbors algorithm, sometimes referred to as KNN or k-NN, is a supervised learning classifier that uses proximity to make classifications or predictions about the grouping of a single data point [7]. Although it can be applied to classification or regression problems, it is commonly employed as a classification algorithm because it relies on the idea that comparable points can be discovered close to one another [8].

The steps of the k-Nearest Neighbors algorithm are [9]:

**Step 1:** Select the number K of the neighbors

The k value in the k-NN algorithm specifies how many neighbors will be examined to determine a particular query point's classification. The example will be placed in the same class as its sole nearest neighbor, for instance, if k=1. In order to avoid either overfitting or underfitting, several values of k must be considered when defining it. Larger values of k may result in strong bias and low variance, while smaller values of k may have high variance but low bias. The selection of k will be heavily influenced by the input data, as data with more outliers or noise will probably perform better with higher values of k. In general, it is advised to choose an odd value for k to prevent classification conflicts.

**Step 2:** Calculate distance between the query example and the current example from the data

Some common distance metrics for k-NN are presented in the figure below.

| **Distance measure** | **Formula** |
|---|---|
| Euclidean | $D = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_n - y_n)^2}$ |
| City block | $D = \lvert x_1 - y_1 \rvert + \lvert x_2 - y_2 \rvert + \ldots\ldots + \lvert x_n - y_n \rvert$ |
| Cosine | $D = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}}$ |
| Correlation | $D = \frac{N\sum xy - (\sum x)(\sum y)}{\sqrt{\left[N\sum x^2 - (\sum x)^2\right]\left[N\sum y^2 - (\sum y)^2\right]}}$ |

Figure 1.1: k-NN distance metrics

**Step 3:** Add the distance and the index of the example to an ordered collection

**Step 4:** Sort the ordered collection of distances and indices from in ascending order by the distances

**Step 5:** Pick the first K entries from the sorted collection

**Step 6:** Get the labels of the selected K entries. Return the mean of the K labels in case of regression or the mode if classification.

## 1.4 Nearest Prototype Classifier

Nearest centroid classifier or nearest prototype classifier is a classification model that assigns to observations the label of the class of training samples whose mean (centroid) is closest to the observation [10].

Given labeled training examples $(x_1, y_1), \ldots, (x_n, y_n)$ with class labels $y_i \in Y$, the per-class centroids are computed as [11]:

$$\mu_l = \frac{1}{m_l} \sum_{i \in C_l} x_i \tag{1.7}$$

where $C_l$ is the set of indices of examples belonging to class $l \in Y$ and m in the length of $C_l$.

The class assigned to an observation $x$ is $y = argmin_{l \in Y} = ||\mu_l - x||$

# Chapter 2

# Implementation Description

## 2.1 Data Exploration

The dataset is loaded using the load_iris() function from sklearn.datasets, as follows:

```
from sklearn import datasets
iris = datasets.load_iris()
```

```
X = iris.data
y = iris.target
```

In order to make it easier to analyse the data, the loaded dataset is converted to a Pandas dataframe:

```
df = pd.DataFrame(data=np.c_[iris['data'], iris['target']],
                  columns=iris['feature_names'] + ['target'])
```

To get more familiar with the data, the first 5 rows of the dataset are printed:

```
df.head()
```

| | sepal length (cm) | sepal width (cm) | petal length (cm) | petal width (cm) | target |
|---|---|---|---|---|---|
| **0** | 5.1 | 3.5 | 1.4 | 0.2 | 0.0 |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 | 0.0 |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 | 0.0 |
| **3** | 4.6 | 3.1 | 1.5 | 0.2 | 0.0 |
| **4** | 5.0 | 3.6 | 1.4 | 0.2 | 0.0 |

It can be observed that the dataset has 4 features: sepal length (cm), sepal width (cm), petal length (cm), petal width (cm).

The names of all possible target classes are also printed:

```
iris['target_names']
```

```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

We have three possible classes: setosa, versicolor and virginica.

df.count() is used to view the number of observations in each class:

```
df.count()
```

```
sepal length (cm)    150
sepal width (cm)     150
petal length (cm)    150
petal width (cm)     150
target               150
dtype: int64
```

Information about the dataset attributes and their data types is obtained with df.info():

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
 4   target             150 non-null    float64
dtypes: float64(5)
memory usage: 6.0 KB
```

As it can be seen, all of the dataset attributes are of type float64.

More information about the data, such as the mean, standard deviation, minimum value and maximum value for each feature can be obtained using df.describe():

```
df.describe()
```

| | sepal length (cm) | sepal width (cm) | petal length (cm) | petal width (cm) | target |
|---|---|---|---|---|---|
| **count** | 150.000000 | 150.000000 | 150.000000 | 150.000000 | 150.000000 |
| **mean** | 5.843333 | 3.057333 | 3.758000 | 1.199333 | 1.000000 |
| **std** | 0.828066 | 0.435866 | 1.765298 | 0.762238 | 0.819232 |
| **min** | 4.300000 | 2.000000 | 1.000000 | 0.100000 | 0.000000 |
| **25%** | 5.100000 | 2.800000 | 1.600000 | 0.300000 | 0.000000 |
| **50%** | 5.800000 | 3.000000 | 4.350000 | 1.300000 | 1.000000 |
| **75%** | 6.400000 | 3.300000 | 5.100000 | 1.800000 | 2.000000 |
| **max** | 7.900000 | 4.400000 | 6.900000 | 2.500000 | 2.000000 |

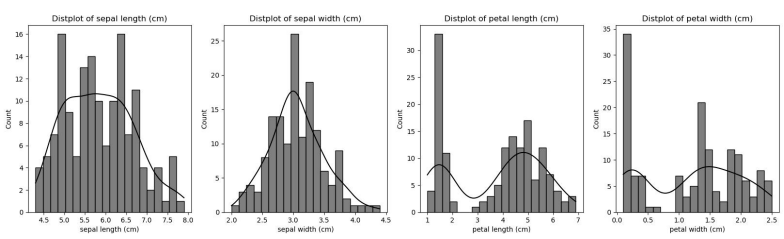
Another useful insight about the dataset is the mean value of each feature, grouped by the species:

```
df.groupby('target').mean()
```

| | sepal length (cm) | sepal width (cm) | petal length (cm) | petal width (cm) |
|---|---|---|---|---|
| **target** | | | | |
| **0.0** | 5.006 | 3.428 | 1.462 | 0.246 |
| **1.0** | 5.936 | 2.770 | 4.260 | 1.326 |
| **2.0** | 6.588 | 2.974 | 5.552 | 2.026 |

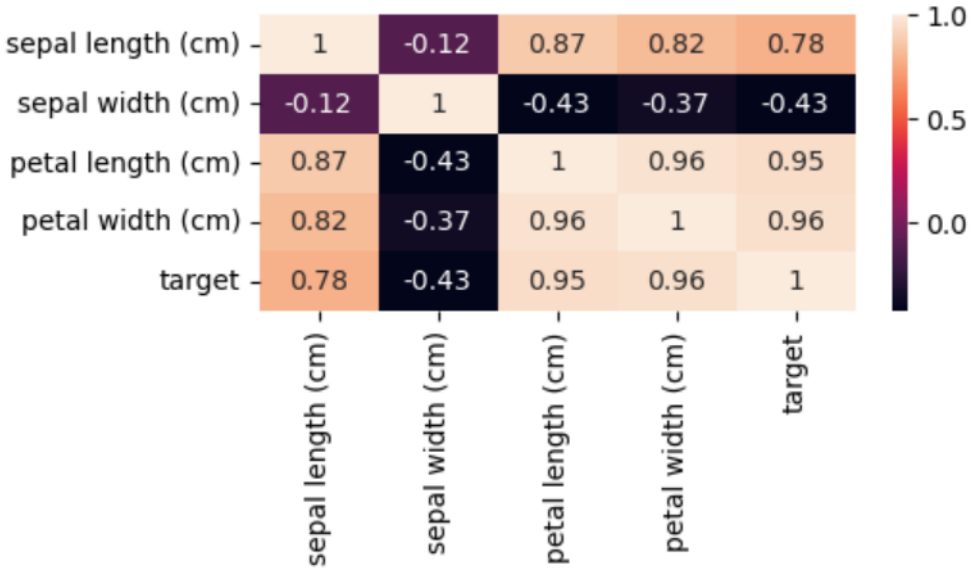
The distribution of values for each attribute can also be represented:

```
# distribution of values for each attribute
plt.figure(1 , figsize = (15 , 6))
for ind, feature in enumerate(iris['feature_names']):
    plt.subplot(1, 4, ind + 1)
    sns.histplot(df[feature] , bins = 20, kde=True, color="black")
    plt.title(f'Distplot of {feature}')
plt.show()
```

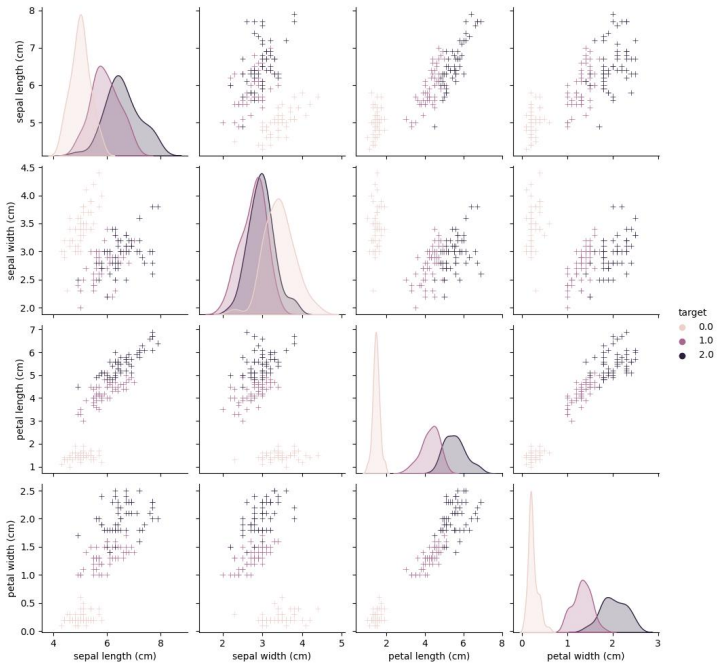


The correlation between features can be observed using correlation heatmaps. As it can be seen below, petal length and petal width seem to be most asssociated.

```
plt.figure(figsize=(5,2))
sns.heatmap(df.corr(), annot=True)
plt.show()
```




```
g = sns.pairplot(df, hue='target', markers='+')
plt.show()
```



## 2.2 Data Preprocessing

The dataset is split into a train set and a test set using the train_test_split function from sklearn.model_selection.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

## 2.3 Principal Component Analysis

The PCA algorithm can be applied to the dataset using the sklearn library, as below:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=n_components)
X_pca_train = pca.fit_transform(X_train)
X_pca_test = pca.transform(X_test)
y_pca_train, y_pca_test = y_train, y_test
```

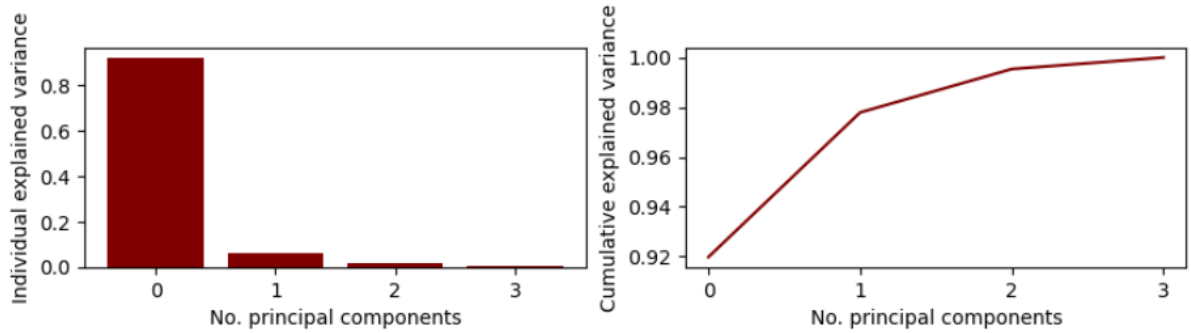
We can also see the computed covariance matrix using the get_covariance() method:

```
pca.get_covariance()
```

```
array([[ 0.06510625, -0.00607287,  0.0716027 ,  0.06978617],
       [-0.00607287,  0.0442848 , -0.02767394, -0.02523075],
       [ 0.0716027 , -0.02767394,  0.10064506,  0.10056317],
       [ 0.06978617, -0.02523075,  0.10056317,  0.10787005]])
```

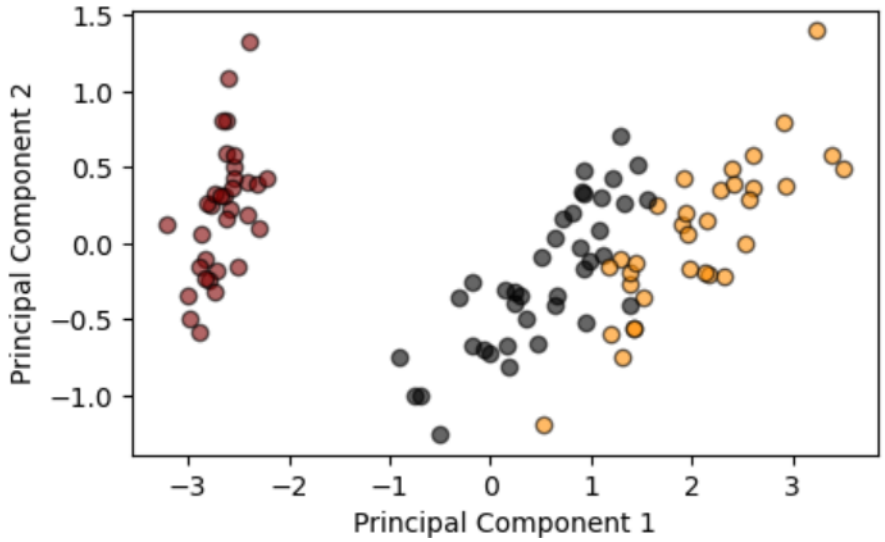
It is also useful to plot the individual explained variance and cumulative explained variance for each of the components:

```
plt.subplot(1, 2, 1)
plt.bar(range(len(explained_variance)), explained_variance, align='center',
        color="maroon")
plt.xlabel('No. principal components')
plt.ylabel('Individual explained variance')
plt.subplot(1, 2, 2)
plt.plot(np.cumsum((explained_variance)), color="maroon")
plt.xlabel('No. principal components')
plt.ylabel('Cumulative explained variance')
```




If we keep only two principal components, we can also plot the data:

```
colors = {0: "maroon", 1: "black", 2: "darkorange"}
plt.figure(figsize=(5, 3))
for X, y in zip(X_pca_train, y_pca_train):
    plt.scatter(X[0], X[1], color=colors[y], edgecolors="black", alpha=0.6)
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()
```




## 2.4 k-Nearest Neighbors

The k-NN algorithm is applied using the sklearn library, in the following manner:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_pca_train, y_pca_train)
```

In the case of 1-NN n_neighbors = 1 and for 3-NN n_neighbors = 3.

The accuracy on the train and test sets is obtained using the score method.

```
knn_score_pca_train = knn.score(X_pca_train, y_pca_train)
knn_score_pca_test = knn.score(X_pca_test, y_pca_test)
```

Some other useful evaluation metrics are the classification matrix and the classification report. In the classification matrix each row represents the instances in an actual class while each column represents the instances in a predicted class, or vice versa. The classification report gives information about important parameters, such as: precision, recall, F1-score, support and accuracy.

```
from sklearn.metrics import plot_confusion_matrix, classification_report
y_pca_pred = knn.predict(X_pca_test)
print(classification_report(y_pca_test, y_pca_pred))
plot_confusion_matrix(knn, X_pca_test, y_pca_test)
plt.show()
```

## 2.5 Nearest Prototype Classifier

The Nearest Prototype Classifier algorithm is applied using the sklearn library, in the following manner:

```
from sklearn.neighbors import NearestCentroid
nc = NearestCentroid()
nc.fit(X_pca_train, y_pca_train)
```

The accuracy on the train and test sets is obtained using the score method.

```
nc_score_train = nc.score(X_pca_train, y_pca_train)
nc_score_test = nc.score(X_pca_test, y_pca_test)
```

The classification report are obtained in a similar manner as before, using the plot_confusion_matrix and classification_report methods from sklearn.metrics.

```
from sklearn.metrics import plot_confusion_matrix, classification_report
y_pca_pred = nc.predict(X_pca_test)
print(classification_report(y_pca_test, y_pca_pred))
plot_confusion_matrix(nc, X_pca_test, y_pca_test)
plt.show()
```

# Chapter 3

# Optimization

A grid search is used to determine which way of splitting the data and distance metric performs best, as seen below.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neighbors import NearestCentroid

model_params = {
    '1KNN' : {
        'model': KNeighborsClassifier(),
        'params': {
            'n_neighbors': [1],
            'metric': ["minkowski", "euclidean", "manhattan", "chebyshev", "cosine", "hamming", "canberra", "braycurtis"]
        }
    },
    '3KNN' : {
        'model': KNeighborsClassifier(),
        'params': {
            'n_neighbors': [3],
            'metric': ["minkowski", "euclidean", "manhattan", "chebyshev", "cosine", "hamming", "canberra", "braycurtis"]
        }
    },
    'NC' : {
        'model': NearestCentroid(),
        'params': {
            'metric': ["minkowski", "euclidean", "manhattan", "chebyshev", "cosine", "hamming", "canberra", "braycurtis"]
        }
    }

}
```

```
from sklearn.model_selection import GridSearchCV
scores = []

for model_name, mp in model_params.items():
    clf = GridSearchCV(mp['model'], mp['params'], cv=20, return_train_score=True, verbose=3)
    clf.fit(X_pca_train, y_pca_train)
    scores.append({
        'model': model_name,
        'best_score': clf.best_score_,
        'best_params': clf.best_params_
    })

df_score = pd.DataFrame(scores,columns=['model','best_score','best_params'])
```

## 3.1 Data Splitting

To figure out which percentage of the data to use for the training set, the grid search was performed, keeping the distance metric as "Minkowski" and observing the training set accuracy for each of the 3 algorithms. The overall accuracy for each of the splitting methods is also calculated. It can be seen that the highest overall accuracy is obtained by using 80% of the data in the training set and 20% of the data in the test set.

| Splitting method | 1-NN | 3-NN | NC | Overall |
|---|---|---|---|---|
| 50% train 50% test | 0.94 | 0.95 | 0.92 | 0.93 |
| 60% train 40% test | 0.91 | 0.92 | 0.92 | 0.91 |
| 70% train 30% test | 0.94 | 0.96 | 0.92 | 0.94 |
| 80% train 20% test | 0.97 | 0.98 | 0.94 | 0.96 |
| 90% train 10% test | 0.95 | 0.95 | 0.91 | 0.93 |

Table 3.1: Accuracy on the train set for each model

## 3.2 Hyperparameter Tuning

The grid search was also used to determine which distance metric performs best for each algorithm in each of the five cases: no PCA, PCA with 1, 2, 3, or 4 components. According to the tables below, the Minkwoski distance performs best in the majority of circumstances.

| Model | Best score | Best distance metric |
|---|---|---|
| 1-NN | 0.950 | minkowski |
| 3-NN | 0.975 | cosine |
| NC | 0.975 | canberra |

Table 3.2: No PCA

| Model | Best score | Best distance metric |
|---|---|---|
| 1-NN | 0.900 | minkowski |
| 3-NN | 0.941 | canberra |
| NC | 0.905 | minkowski |

Table 3.3: PCA with 1 principal component

| Model | Best score | Best Distance metric |
|---|---|---|
| 1-NN | 0.933 | minkowski |
| 3-NN | 0.958 | minkowski |
| NC | 0.908 | minkowski |

Table 3.4: PCA with 2 principal components

| Model | Best score | Best distance metric |
|---|---|---|
| 1-NN | 0.967 | manhattan |
| 3-NN | 0.967 | minkowski |
| NC | 0.925 | minkowski |

Table 3.5: PCA with 3 principal components

| Model | Best score | Best distance metric |
|---|---|---|
| 1-NN | 0.958 | minkowski |
| 3-NN | 0.975 | braycurtis |
| NC | 0.930 | minkowski |

Table 3.6: PCA with 4 principal components

# Chapter 4

# Results

## 4.1 Results

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 12 |
| 1 | 0.82 | 1.00 | 0.90 | 9 |
| 2 | 1.00 | 0.78 | 0.88 | 9 |
| accuracy | | | 0.93 | 30 |
| macro avg | 0.94 | 0.93 | 0.92 | 30 |
| weighted avg | 0.95 | 0.93 | 0.93 | 30 |

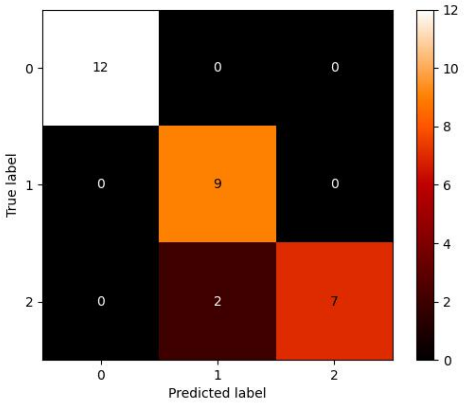



Figure 4.1: k-Nearest Neighbors, k=1, no PCA

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 12 |
| 1 | 1.00 | 1.00 | 1.00 | 9 |
| 2 | 1.00 | 1.00 | 1.00 | 9 |
| accuracy | | | 1.00 | 30 |
| macro avg | 1.00 | 1.00 | 1.00 | 30 |
| weighted avg | 1.00 | 1.00 | 1.00 | 30 |

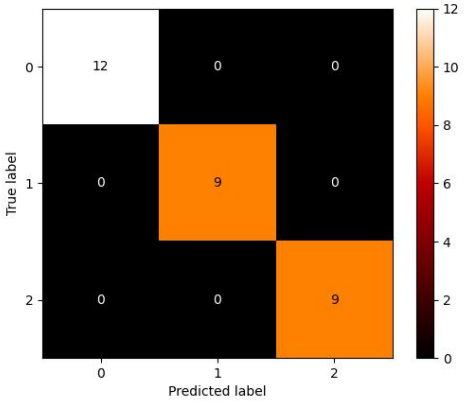



Figure 4.2: k-Nearest Neighbors, k=3, no PCA

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 12 |
| 1 | 0.82 | 1.00 | 0.90 | 9 |
| 2 | 1.00 | 0.78 | 0.88 | 9 |
| accuracy | | | 0.93 | 30 |
| macro avg | 0.94 | 0.93 | 0.92 | 30 |
| weighted avg | 0.95 | 0.93 | 0.93 | 30 |

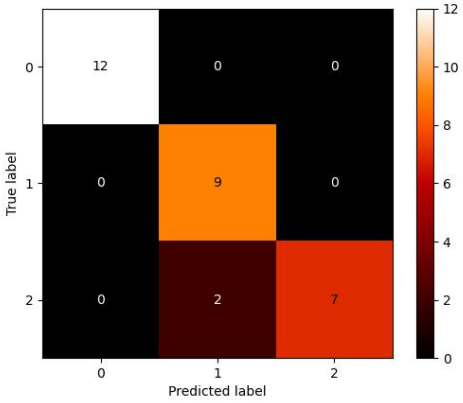



Figure 4.3: Nearest Prototype, no PCA

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 8 |
| 1 | 1.00 | 0.62 | 0.76 | 13 |
| 2 | 0.64 | 1.00 | 0.78 | 9 |
| accuracy | | | 0.83 | 30 |
| macro avg | 0.88 | 0.87 | 0.85 | 30 |
| weighted avg | 0.89 | 0.83 | 0.83 | 30 |

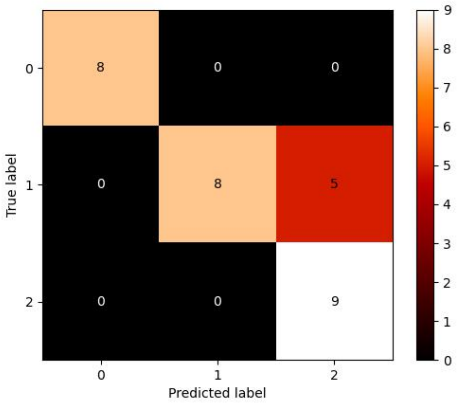



Figure 4.4: k-Nearest Neighbors, k=1, 1 principal component

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 8 |
| 1 | 1.00 | 0.77 | 0.87 | 13 |
| 2 | 0.75 | 1.00 | 0.86 | 9 |
| accuracy | | | 0.90 | 30 |
| macro avg | 0.92 | 0.92 | 0.91 | 30 |
| weighted avg | 0.93 | 0.90 | 0.90 | 30 |

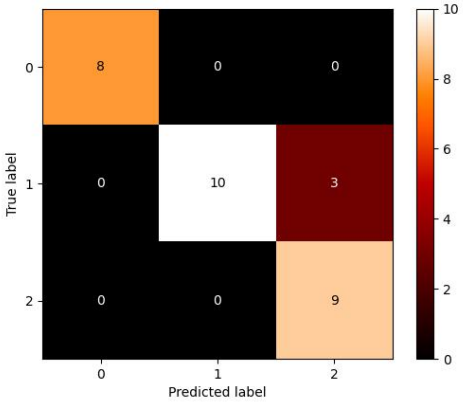



Figure 4.5: k-Nearest Neighbors, k=3, 1 principal component

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 8 |
| 1 | 1.00 | 0.77 | 0.87 | 13 |
| 2 | 0.75 | 1.00 | 0.86 | 9 |
| accuracy | | | 0.90 | 30 |
| macro avg | 0.92 | 0.92 | 0.91 | 30 |
| weighted avg | 0.93 | 0.90 | 0.90 | 30 |

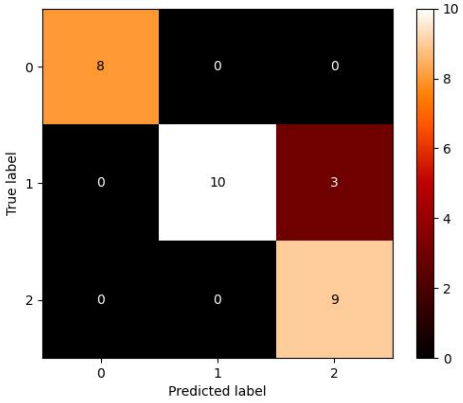



Figure 4.6: Nearest Prototype, 1 principal component

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 7 |
| 1 | 0.90 | 0.90 | 0.90 | 10 |
| 2 | 0.92 | 0.92 | 0.92 | 13 |
| accuracy | | | 0.93 | 30 |
| macro avg | 0.94 | 0.94 | 0.94 | 30 |
| weighted avg | 0.93 | 0.93 | 0.93 | 30 |

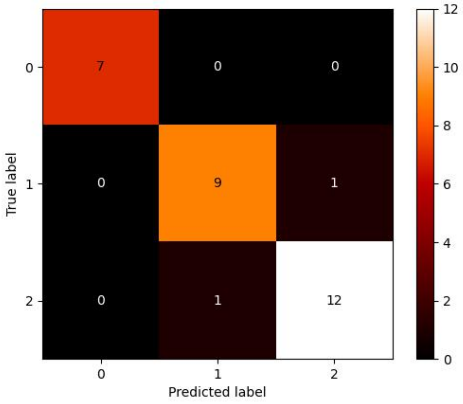



Figure 4.7: k-Nearest Neighbors, k=1, 2 principal components

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 7 |
| 1 | 0.91 | 1.00 | 0.95 | 10 |
| 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 |

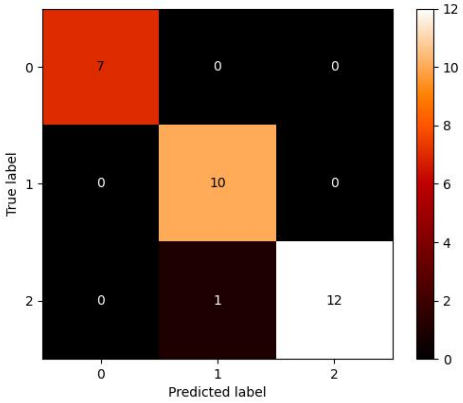



Figure 4.8: k-Nearest Neighbors, k=3, 2 principal components

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 7 |
| 1 | 0.83 | 1.00 | 0.91 | 10 |
| 2 | 1.00 | 0.85 | 0.92 | 13 |
| accuracy | | | 0.93 | 30 |
| macro avg | 0.94 | 0.95 | 0.94 | 30 |
| weighted avg | 0.94 | 0.93 | 0.93 | 30 |

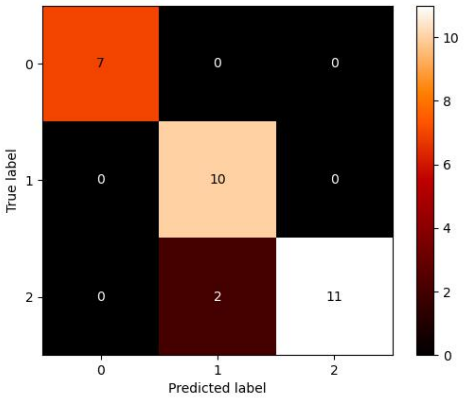



Figure 4.9: Nearest Prototype, 2 principal components

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 11 |
| 1 | 0.92 | 1.00 | 0.96 | 11 |
| 2 | 1.00 | 0.88 | 0.93 | 8 |
| accuracy | | | 0.97 | 30 |
| macro avg | 0.97 | 0.96 | 0.96 | 30 |
| weighted avg | 0.97 | 0.97 | 0.97 | 30 |

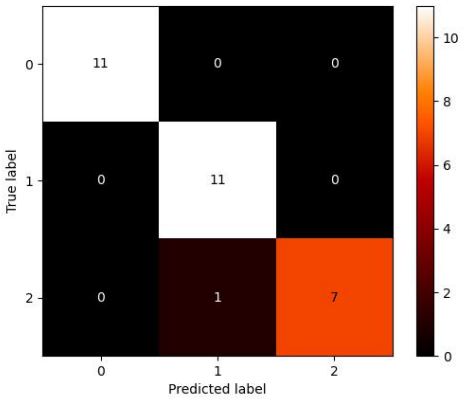



Figure 4.10: k-Nearest Neighbors, k=1, 3 principal components

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 11 |
| 1 | 0.92 | 1.00 | 0.96 | 11 |
| 2 | 1.00 | 0.88 | 0.93 | 8 |
| accuracy | | | 0.97 | 30 |
| macro avg | 0.97 | 0.96 | 0.96 | 30 |
| weighted avg | 0.97 | 0.97 | 0.97 | 30 |

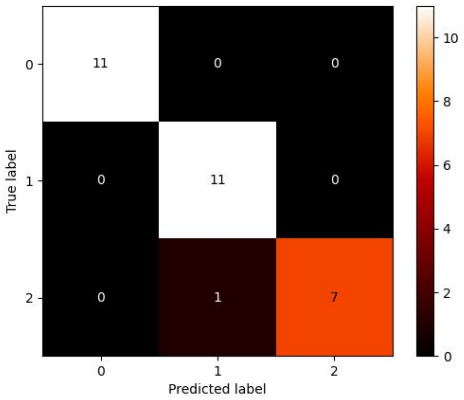



Figure 4.11: k-Nearest Neighbors, k=3, 3 principal components

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 11 |
| 1 | 0.85 | 1.00 | 0.92 | 11 |
| 2 | 1.00 | 0.75 | 0.86 | 8 |
| accuracy | | | 0.93 | 30 |
| macro avg | 0.95 | 0.92 | 0.92 | 30 |
| weighted avg | 0.94 | 0.93 | 0.93 | 30 |

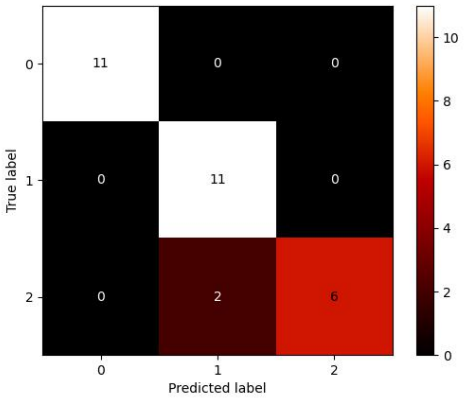



Figure 4.12: Nearest Prototype, 3 principal components

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 8 |
| 1 | 1.00 | 0.89 | 0.94 | 9 |
| 2 | 0.93 | 1.00 | 0.96 | 13 |
| accuracy | | | 0.97 | 30 |
| macro avg | 0.98 | 0.96 | 0.97 | 30 |
| weighted avg | 0.97 | 0.97 | 0.97 | 30 |

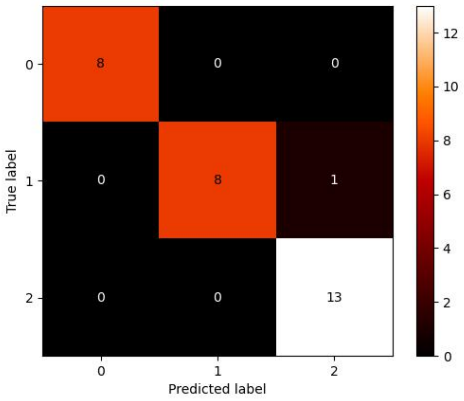



Figure 4.13: k-Nearest Neighbors, k=1, 4 principal components

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 8 |
| 1 | 1.00 | 0.89 | 0.94 | 9 |
| 2 | 0.93 | 1.00 | 0.96 | 13 |
| accuracy | | | 0.97 | 30 |
| macro avg | 0.98 | 0.96 | 0.97 | 30 |
| weighted avg | 0.97 | 0.97 | 0.97 | 30 |

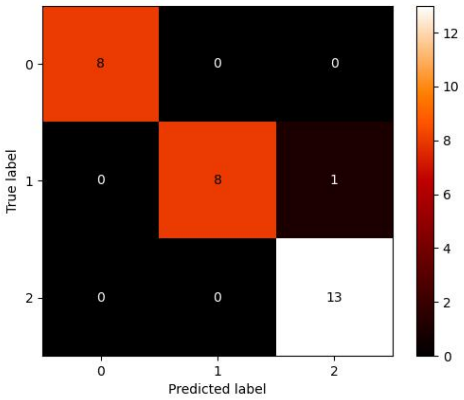



Figure 4.14: k-Nearest Neighbors, k=3, 4 principal components

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 8 |
| 1 | 1.00 | 0.56 | 0.71 | 9 |
| 2 | 0.76 | 1.00 | 0.87 | 13 |
| accuracy | | | 0.87 | 30 |
| macro avg | 0.92 | 0.85 | 0.86 | 30 |
| weighted avg | 0.90 | 0.87 | 0.86 | 30 |

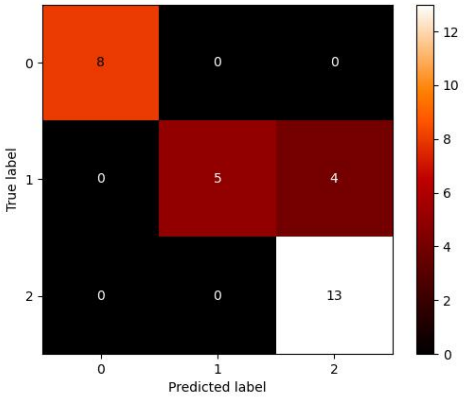



Figure 4.15: Nearest Prototype, 4 principal components

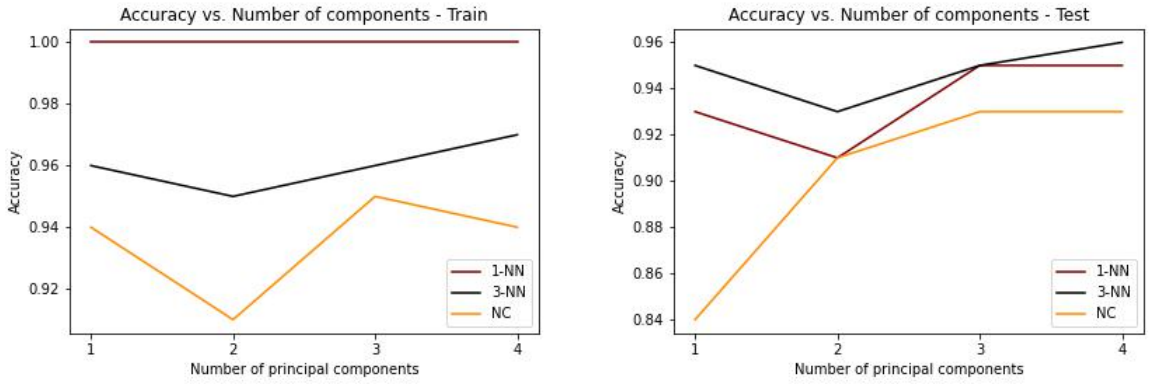
## 4.2 Comparison

| | No PCA | 1 component | 2 components | 3 components | 4 components |
|---|---|---|---|---|---|
| 1-NN | 1 | 1 | 1 | 1 | 1 |
| 3-NN | 0.96 | 0.96 | 0.95 | 0.96 | 0.97 |
| NC | 0.94 | 0.94 | 0.91 | 0.95 | 0.94 |

Table 4.1: Train set accuracies

| | No PCA | 1 component | 2 components | 3 components | 4 components |
|---|---|---|---|---|---|
| 1-NN | 0.93 | 0.93 | 0.91 | 0.95 | 0.95 |
| 3-NN | 0.93 | 0.95 | 0.93 | 0.95 | 0.96 |
| NC | 0.93 | 0.84 | 0.91 | 0.93 | 0.93 |

Table 4.2: Test set accuracies




According to the tables above, the best accuracy on the test set (0.96) is obtained for 3-NN with 4 components. Additionally, it can be seen that accuracy tends increase with the number of components kept, regardless of the algorithm used. This makes sense because the quantity of information preserved is directly proportional to the number

of principal components. However, the size of the dataset is reduced less when more principal components are kept, which leads to longer computation time and the need for more processing power in the case of big datasets.

When it comes to the k-NN algorithm, we notice that the performance on the training set is better when k = 1 than when k = 3. However, on the test set, the performance is better for 3-NN. In comparison, the Nearest Centroid algorithm performs worse than k-NN both on the training set and on the test set.

# Chapter 5

# Conclusions

## 5.1 Principal Component Analysis

- The principal component analysis algorithm can be used to visualize data with more than two dimensions. The multidimensional data is projected onto a two-dimensional space (only the first two primary components are retained), and then it can be plotted normally.
- Another important use case of principal component analysis is dimensionality reduction of the dataset. This is especially useful when dealing with large datasets, as it can significantly decrease computation time and the amount of required processing power.

## 5.2 k-Nearest Neighbors

- Not a linear classifier.
- More complex and has a tuning parameter k that needs to be set by the user.
- Slow at classification time.
- High memory usage for storing all data points.
- Robust to data outliers.

## 5.3 Nearest Centroid

- Linear classifier.
- Simple algoritm which doesn't require hyperparameter tuning.
- Fast at classification time.
- Requires only little memory, as it is only storing the centroids.
- Robust to mislabeled data.
- Robust to class imbalance.

# Bibliography


[1] Mitchell T.M. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997.

[2] Alpaydin E. *Introduction to Machine Learning*. MIT press, 2020.

[3] What is Machine Learning? `https://developers.google.com/machine-learning/intro-to-ml/what-is-ml`. Accessed: 2022-06-11.

[4] Abdi H. and Williams L. J. Principal Component Analysis. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(4):433–459, 2010.

[5] Neagoe V. Computational Intelligence 1 Laboratory Platforms. 2022.

[6] Principal Component Analysis. `https://en.wikipedia.org/wiki/Principal_component_analysis`. Accessed: 2023-01-04.

[7] Peterson L. E. K-Nearest Neighbor. *Scholarpedia*, 4(2):1883, 2009.

[8] K-Nearest Neighbors Algorithm. `https://www.ibm.com/topics/knn`. Accessed: 2023-01-04.

[9] Kramer O. K-Nearest Neighbors. In *Dimensionality reduction with unsupervised nearest neighbors*, pages 13–23. Springer, 2013.

[10] Gou J. et al. A local mean-based k-nearest centroid neighbor classifier. *The Computer Journal*, 55(9):1058–1071, 2012.

[11] Nearest Centroid Classifier. `https://en.wikipedia.org/wiki/Nearest_centroid_classifier`. Accessed: 2023-01-04.

# Appendix - Code

## Import libraries

In [1]:

```
import matplotlib.pyplot as plt
plt.rcParams.update(plt.rcParamsDefault)
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
```

---

## Load Iris Dataset

In [2]:

```
from sklearn import datasets
iris = datasets.load_iris()
```

In [3]:

```
X = iris.data
y = iris.target
```

---

## Split into training and test dataset

In [4]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

---

## PCA

### Applying PCA

In [5]:

```
n_components=4
```

In [6]:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=n_components)
X_pca_train = pca.fit_transform(X_train)
X_pca_test = pca.transform(X_test)
y_pca_train, y_pca_test = y_train, y_test
```

## Analysing PCA Results

In [7]:

```
if n_components==2:
    colors = {0: "maroon", 1: "black", 2: "darkorange"}
    plt.figure(figsize=(5, 3))
    for X, y in zip(X_pca_train, y_pca_train):
        plt.scatter(X[0], X[1], color=colors[y], edgecolors="black", alpha=0.6)
    plt.xlabel("Principal Component 1")
    plt.ylabel("Principal Component 2")
    plt.show()
```

In [8]:

```
pca.get_covariance()
```

Out[8]:

```
array([[ 0.60335504, -0.04913866,  1.14694118,  0.47295378],
       [-0.04913866,  0.2047479 , -0.37621849, -0.14415966],
       [ 1.14694118, -0.37621849,  2.97192157,  1.24180392],
       [ 0.47295378, -0.14415966,  1.24180392,  0.5575098 ]])
```

In [9]:

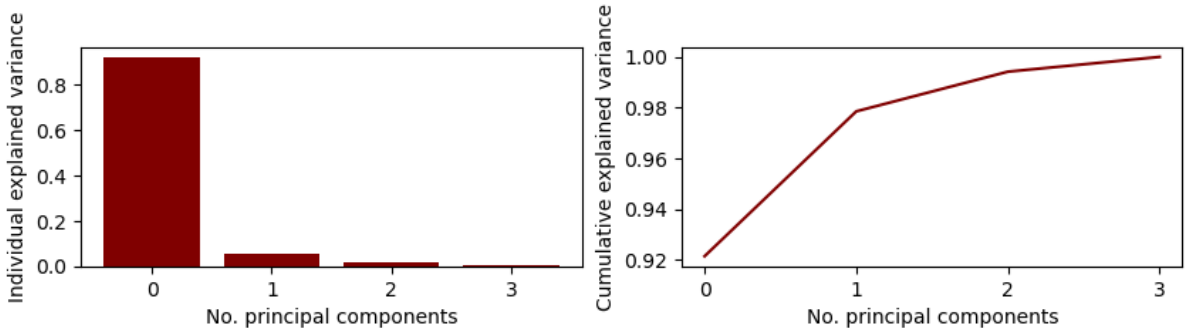
```
explained_variance = pca.explained_variance_ratio_
explained_variance
```

Out[9]:

```
array([0.92144673, 0.05709714, 0.01564435, 0.00581179])
```

In [10]:

```
plt.figure(figsize=(10, 2))
plt.subplot(1, 2, 1)
plt.bar(range(len(explained_variance)), explained_variance, align='center',
        color="maroon")
plt.ylabel('Individual explained variance')
plt.xlabel('No. principal components')
plt.xticks(list(range(len(explained_variance))))
plt.subplot(1, 2, 2)
plt.plot(np.cumsum((explained_variance)), color="maroon")
plt.xticks(list(range(len(explained_variance))))
plt.ylabel('Cumulative explained variance')
plt.xlabel('No. principal components')
plt.show()
```



## Classification

### kNN (k=1)

In [11]:

```
from sklearn.neighbors import KNeighborsClassifier
knn1 = KNeighborsClassifier(n_neighbors=1)
knn1.fit(X_pca_train, y_pca_train)
```

Out[11]:

```
KNeighborsClassifier(n_neighbors=1)
```

In [12]:

```
knn1_score_pca_train = knn1.score(X_pca_train, y_pca_train)
knn1_score_pca_test = knn1.score(X_pca_test, y_pca_test)
```

In [13]:

```
print(f"Train score with PCA: {knn1_score_pca_train}")
print(f"Test score with PCA: {knn1_score_pca_test}")
```

```
Train score with PCA: 1.0
Test score with PCA: 0.9666666666666667
```

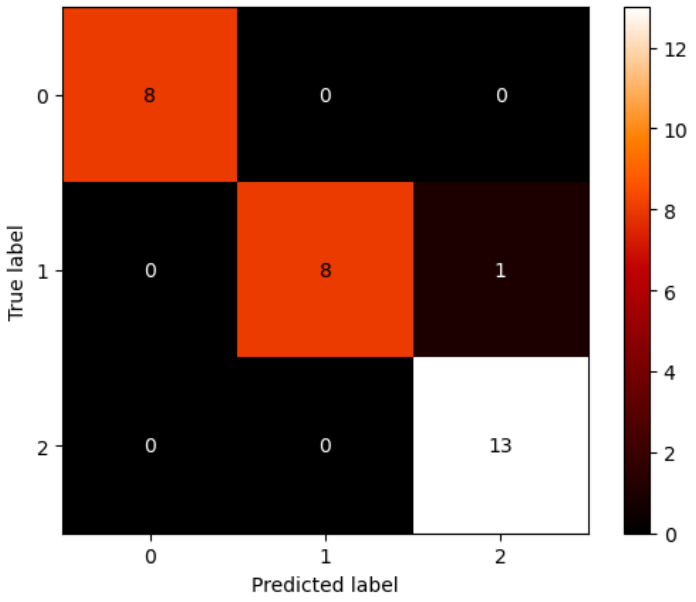
In [14]:

```
from sklearn.metrics import plot_confusion_matrix, classification_report
y_pca_pred = knn1.predict(X_pca_test)
print(classification_report(y_pca_test, y_pca_pred))
plot_confusion_matrix(knn1, X_pca_test, y_pca_test, cmap="gist_heat")
# plt.savefig(f"poze/result_1knn_{n_components}comp_classification_matrix.jpg")
plt.show()
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         8
           1       1.00      0.89      0.94         9
           2       0.93      1.00      0.96        13

    accuracy                           0.97        30
   macro avg       0.98      0.96      0.97        30
weighted avg       0.97      0.97      0.97        30
```



### kNN (k=3)

In [15]:

```
from sklearn.neighbors import KNeighborsClassifier
knn3 = KNeighborsClassifier(3)
knn3.fit(X_pca_train, y_pca_train)
```

Out[15]:

```
KNeighborsClassifier(n_neighbors=3)
```

In [16]:

```
knn3_score_pca_train = knn3.score(X_pca_train, y_pca_train)
knn3_score_pca_test = knn3.score(X_pca_test, y_pca_test)
```

In [17]:

```
print(f"Train score with PCA: {knn3_score_pca_train}")
print(f"Test score with PCA: {knn3_score_pca_test}")
```

```
Train score with PCA: 0.9583333333333334
Test score with PCA: 0.9666666666666667
```

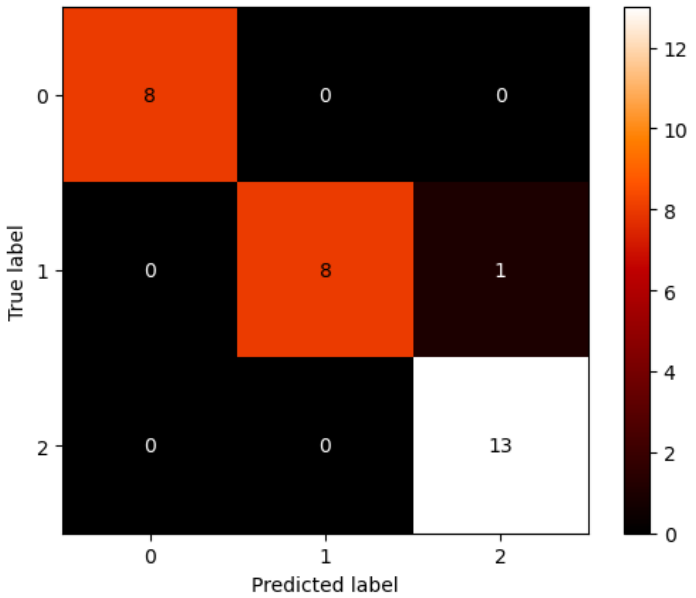
In [18]:

```
from sklearn.metrics import plot_confusion_matrix, classification_report
y_pca_pred = knn3.predict(X_pca_test)
print(classification_report(y_pca_test, y_pca_pred))
plot_confusion_matrix(knn3, X_pca_test, y_pca_test, cmap="gist_heat")
# plt.savefig(f"poze/result_3knn_{n_components}comp_classification_matrix.jpg")
plt.show()
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         8
           1       1.00      0.89      0.94         9
           2       0.93      1.00      0.96        13

    accuracy                           0.97        30
   macro avg       0.98      0.96      0.97        30
weighted avg       0.97      0.97      0.97        30
```



## Supervised k-Means

In [19]:

```
from sklearn.neighbors import NearestCentroid
nc = NearestCentroid()
nc.fit(X_pca_train, y_pca_train)
```

Out[19]:

```
NearestCentroid()
```

In [20]:

```
nc_score_train = nc.score(X_pca_train, y_pca_train)
nc_score_test = nc.score(X_pca_test, y_pca_test)
```

In [21]:

```
print(f"Train score with PCA: {nc_score_train}")
print(f"Test score with PCA: {nc_score_test}")
```

```
Train score with PCA: 0.9416666666666667
Test score with PCA: 0.8666666666666667
```

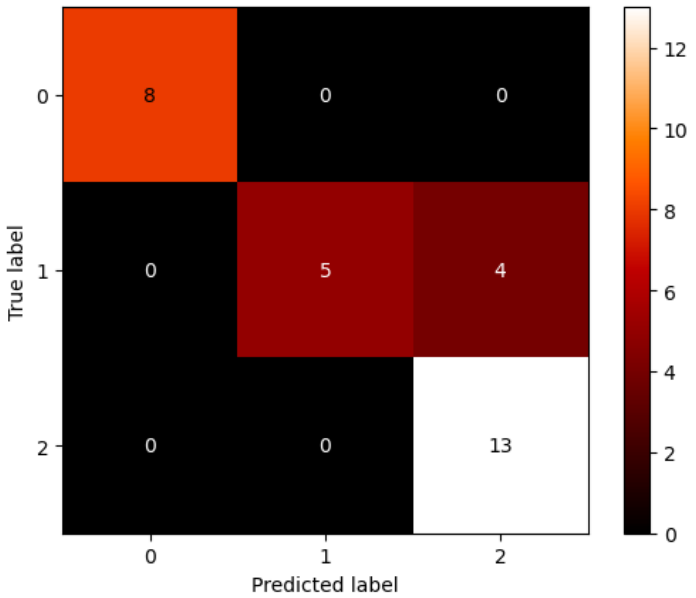
In [22]:

```
from sklearn.metrics import plot_confusion_matrix, classification_report
y_pca_pred = nc.predict(X_pca_test)
print(classification_report(y_pca_test, y_pca_pred))
plot_confusion_matrix(nc, X_pca_test, y_pca_test, cmap="gist_heat")
# plt.savefig(f"poze/result_nc_{n_components}comp_classification_matrix.jpg")
plt.show()
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         8
           1       1.00      0.56      0.71         9
           2       0.76      1.00      0.87        13

    accuracy                           0.87        30
   macro avg       0.92      0.85      0.86        30
weighted avg       0.90      0.87      0.86        30
```



## GridSearch

In [24]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neighbors import NearestCentroid

model_params = {
    '1KNN' : {
        'model': KNeighborsClassifier(),
        'params': {
            'n_neighbors': [1],
            'metric': ["minkowski", "euclidean", "manhattan", "chebyshev", "cosine", "hamming", "canberra", "braycurtis"]
        }
    },
    '3KNN' : {
        'model': KNeighborsClassifier(),
        'params': {
            'n_neighbors': [3],
            'metric': ["minkowski", "euclidean", "manhattan", "chebyshev", "cosine", "hamming", "canberra", "braycurtis"]
        }
    },
    'NC' : {
        'model': NearestCentroid(),
        'params': {
            'metric': ["minkowski", "euclidean", "manhattan", "chebyshev", "cosine", "hamming", "canberra", "braycurtis"]
        }
    }

}
```

In [25]:

```
from sklearn.model_selection import GridSearchCV
scores = []

for model_name, mp in model_params.items():
    clf = GridSearchCV(mp['model'], mp['params'], cv=20, return_train_score=True, verbose=3)
    clf.fit(X_train, y_train)
    scores.append({
        'model': model_name,
        'best_score': clf.best_score_,
        'best_params': clf.best_params_
    })

df_score = pd.DataFrame(scores,columns=['model','best_score','best_params'])
```

```
Fitting 20 folds for each of 8 candidates, totalling 160 fits
[CV 1/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 2/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 3/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 4/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=0.833) total time=   0.0s
[CV 5/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 6/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=0.833) total time=   0.0s
[CV 7/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 8/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 9/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 10/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 11/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 12/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 13/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 14/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=0.833) total time=   0.0s
[CV 15/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 16/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 17/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
[CV 18/20] END metric=minkowski, n_neighbors=1;, score=(train=1.000, test=1.000) total time=   0.0s
```

In [26]:

```
df_score = pd.DataFrame(scores,columns=['model','best_score','best_params'])
print(df_score)
```

```
  model  best_score                                best_params
0  1KNN    0.958333  {'metric': 'minkowski', 'n_neighbors': 1}
1  3KNN    0.975000     {'metric': 'cosine', 'n_neighbors': 3}
2    NC    0.975000                      {'metric': 'cosine'}
```