People's Democratic Republic of Algeria
University of Science and Technology HOUARI BOUMEDIENE
Faculty of Computer Science

Data Mining Report on Supervised and Unsupervised Algorithms

Written by

- Mouhoub Massinissa

Contents

Introduction

Data mining is an essential discipline in the field of data science. It consists of extracting useful knowledge from large datasets, often complex and heterogeneous. The main objective of *data mining* is to discover patterns, relationships, or trends that can guide decision-making in various domains, including healthcare, finance, marketing, and many other sectors. In the context of this report, we explore two fundamental approaches in *data mining*: supervised algorithms and unsupervised algorithms. These two types of algorithms play a crucial role in data processing and analysis, each having distinct objectives and methods. Supervised algorithms rely on labeled datasets, allowing the prediction of labels or values for new data. Unsupervised algorithms, on the other hand, work on unlabeled data and focus on identifying inherent structures or groupings. This report aims to present a comprehensive analysis of several supervised and unsupervised algorithms, with an emphasis on their definitions, operating principles, and practical applications. An essential part of this study consists of evaluating the performance of these algorithms on different datasets and optimizing their hyperparameters. Thus, this report is structured around the following elements:

1. Theoretical definitions.
2. Source code used.
3. Hyperparameter optimization for each algorithm.

Chapter 1

Theoretical Definition of Classification Algorithms

1.1 Definition of Unsupervised Classification

Unsupervised classification, or clustering, is a machine learning method where the goal is to group unlabeled data into homogeneous sets. The objective is to discover hidden structures or intrinsic patterns in the data, by identifying groups or clusters of similar points.

1.2 Definition of Supervised Classification

Supervised classification is a machine learning method where a model is trained on a labeled dataset to predict the labels of new data. The model learns the relationship between input features and output labels, enabling predictions on unknown data.

1.3 Comparison Between Unsupervised and Supervised Classification

The comparison between the two approaches can be summarized in the following table:

1.3.1 Choice of Approach

The choice between a supervised or unsupervised approach depends strongly on the problem to be solved and the availability of data:

Aspect	**Unsupervised Classification**	**Supervised Classification**
Required data	Unlabeled	Labeled
Objective	Structure discovery	Label prediction
Typical applications	Data exploration, segmentation	Recognition, predictive classification
Algorithm examples	K-means, DBSCAN	KNN, Decision Trees, SVM

Table 1.1: Comparison between unsupervised and supervised classification.

- If labels are available, supervised classification is generally the best option for predictive tasks.
- In the absence of labels, unsupervised classification can be useful for exploring data or identifying hidden patterns.

Chapter 2

Source Code Used

In this section, we will explain the key elements of the code used to obtain the presented results. This explanation will also serve as a guide to better understand how the software works.

We used functions already implemented in widely recognized and adopted libraries on the market, such as **scikit-learn** and **TensorFlow**. These libraries benefit from an excellent reputation and a very large community of developers, which guarantees their reliability and robustness.

We will begin by presenting the functions used to run the classification algorithms.

2.1 Unsupervised Classification Algorithms

In this section, we address the unsupervised classification algorithms, which allow identifying groupings in the data without requiring prior labels.

As a performance measure, we worked with the silhouette score, which measures the quality of the clustering by evaluating how well data points are assigned to their respective clusters while being distinct from other clusters.

The silhouette score is computed for each data point by taking into account two main values:

- **Intra-cluster cohesion** (a): the average distance between a point and the other points in its own cluster.
- **Inter-cluster separation** (b): the average distance between a point and the points in the nearest cluster to which it does not belong.

The silhouette score formula for a point is given by:

$$s = \frac{b - a}{\max(a, b)}$$

where s ranges from -1 to 1. A score close to 1 indicates that the point is well assigned to its cluster, while a score close to -1 suggests a poor assignment.

In our study, the silhouette score was used to evaluate and compare the performance of the different unsupervised classification algorithms on various datasets.

In order to facilitate the management of the **pandas** dataframe, we created a class that automates many things:

```
1  from pandas import DataFrame
2  from pandas.api.types import is_numeric_dtype
3  from pandas import read_csv, concat
4  from sklearn.preprocessing import OneHotEncoder, StandardScaler,
       LabelEncoder
5  from scipy.io.arff import loadarff
6  from sklearn.model_selection import train_test_split
7  import numpy as np
8
9  class DF:
10     def __init__(self, df=None, X_train=None, X_test=None,
           y_train=None, y_test=None, type=None):
11         self.df = df
12         self.X_train = X_train
13         self.X_test = X_test
14         self.y_train = y_train
15         self.y_test = y_test
16         self.type = type
17
18     def reading_data(self, filepath, file_type='csv'):
19         if file_type == 'csv':
20             self.df = read_csv(filepath, na_values=['?', 'null',
                   'NaN'])
21             self.type = 'csv'
22         elif file_type == 'arff':
23             data = loadarff(filepath)
24             self.df = DataFrame(data[0])
25             self.df = self.df.map(lambda x: x.decode('utf-8') if
                   isinstance(x, bytes) else x)
26             self.type='arff'
27         else:
28             raise Exception('Type du fichier non support')
29
30
```

```
31      def preprocessing(self, exclude=[], scalling_method='standard')
            -> DataFrame:
32          print("\n\nLe prtraitement a commenc!\n\n")
33
34          print("Remplissage des valeurs nules..\n")
35          self.__missing_values()
36          print("Remplissage termin!\n")
37
38          print("Normalisation des donnes..\n")
39          self.__scalling_data(method=scalling_method,
                exclude=exclude)
40
41          print("Encodage des donnes..\n")
42          self.__encoding_data(exclude)
43          print("Encodage termin!")
44
45
46      def encoding_class(self, target_column):
47          target_encoder = LabelEncoder()
48          self.df[target_column] =
                target_encoder.fit_transform(self.df[target_column])
49
50      def splitting_data(self, target_column):
51          X = self.df.drop(target_column, axis=1)
52          y = self.df[target_column]
53
54          self.X_train, self.X_test, self.y_train, self.y_test =
                train_test_split(X, y, test_size=0.2)
55
56      def head(self):
57          print(self.df.head())
58
59      def drop(self, columns=[]):
60          self.df = self.df.drop(columns, axis=1)
61
62      # Private methods
63      def __missing_values(self):
64          if self.type == 'csv':
65              for column in self.df.columns:
66                  if is_numeric_dtype(self.df[column]):
67                      self.df[column] =
                            self.df[column].fillna(self.df[column].mean())
68                  else:
```

```
69                      self.df[column] =
                            self.df[column].fillna(self.df[column].mode()[0])
70          elif self.type == 'arff':
71              for column in self.df.columns:
72                  if is_numeric_dtype(self.df[column]):
73                      self.df[column] = self.df[column].replace({'?':
                            np.nan}).astype(float)
74                      self.df[column] =
                            self.df[column].fillna(self.df[column].mean())
75                  else:
76                      self.df[column] = self.df[column].replace({'?':
                            np.nan})
77                      self.df[column] =
                            self.df[column].fillna(self.df[column].mode()[0])
78          else:
79              raise Exception('Mthode non  implment  encore!')
80
81
82      def __encoding_data(self, exclude):
83          object_columns =
                self.df.select_dtypes(exclude=['number']).columns.tolist()
84
85          if object_columns:
86              ohe =
                    OneHotEncoder(sparse_output=False).set_output(transform='pandas')
87              if exclude:
88                  for col in exclude:
89                      if col in object_columns:
90                          object_columns.remove(col)
91
92              encoded_data = ohe.fit_transform(self.df[object_columns])
93              self.df = concat([self.df, encoded_data],
                    axis=1).drop(columns=object_columns)
94          else:
95              return
96
97      def __scalling_data(self, exclude, method='standard'):
98          numeric_columns =
                self.df.select_dtypes(include=['number']).columns.tolist()
99
100         if numeric_columns:
101             if method == 'standard':
102                 scaler = StandardScaler()
```

```
103
104                 if exclude:
105                     for col in exclude:
106                         if col in numeric_columns:
107                             numeric_columns.remove(col)
108
109                 self.df[numeric_columns] =
                        scaler.fit_transform(self.df[numeric_columns])
110
111             else:
112                 raise Exception('Mthode non  implment  encore!')
113         else:
114             return
```

Listing 2.1: Classe du dataframe

2.1.1 K-means

The K-means algorithm is a data partitioning algorithm that divides points into K clusters by minimizing intra-cluster variance. The implementation we used is based on the **scikit-learn** library, which provides an optimized and easy-to-use method for clustering 2.1. The main process is as follows:

1. Initialization of the K centroids randomly or using a method such as `k-means++`.
2. Assignment of each data point to the nearest centroid.
3. Update of centroid positions by computing the mean of the points assigned to each cluster.
4. Repetition of steps 2 and 3 until convergence (i.e., when the centroids no longer change significantly or a maximum number of iterations is reached).

Source code used:

```
1 from sklearn.cluster import KMeans
2 from sklearn.metrics import silhouette_score
3
4 def kmeans(df, nb_clusters=None, auto=False):
5     if not auto:
6         clusterer = KMeans(n_clusters=nb_clusters)
```

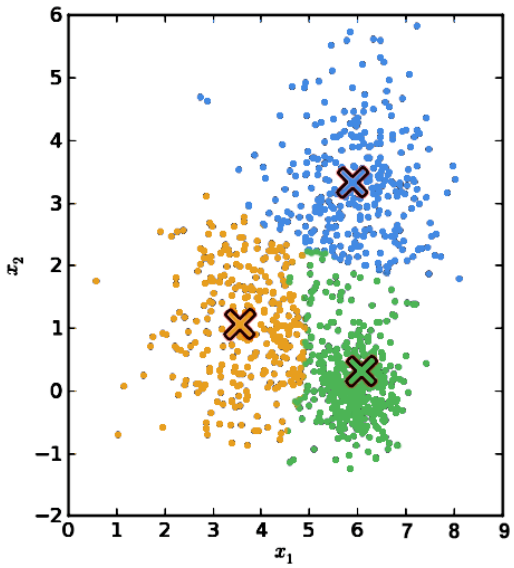

Figure 2.1: Kmeans

```
7           labels = clusterer.fit_predict(df)
8
9           silouhette = silhouette_score(df, labels)
10
11          return {
12              'algorithm': 'kmeans',
13              'type': 'unsupervised',
14              'silouhette': silouhette,
15              'k': nb_clusters,
16          }
17      else:
18          max_silouhette = 0
19          max_k = 0
20          for i in range(2, 30):
21              clusterer = KMeans(n_clusters=i)
22              labels = clusterer.fit_predict(df)
23
24              silouhette = silhouette_score(df, labels)
25
26              if silouhette > max_silouhette:
27                  max_silouhette = silouhette
28                  max_k = i
29
30
```

```
31          return {
32              'algorithm': 'kmeans',
33              'type': 'unsupervised',
34              'silouhette': float(max_silouhette),
35              'k': max_k,
36          }
```

Listing 2.2: Implémentation de K-means avec scikit-learn

This function accepts 3 parameters:

- `df` The **pandas** dataframe.
- `nb_clusters` Corresponds to the number of clusters.
- `auto` Allows automating the algorithm in order to obtain the best parameter to maximize performance.

2.1.2 K-medoids

The K-medoids algorithm is a robust variant of K-means that uses actual data points as centroids, thus reducing the impact of outliers. The implementation is based on the **scikit-learn-extra** library, offering an efficient method for partitioning. The main process is similar to K-means 2.2, but the centroids are chosen from among the existing points.

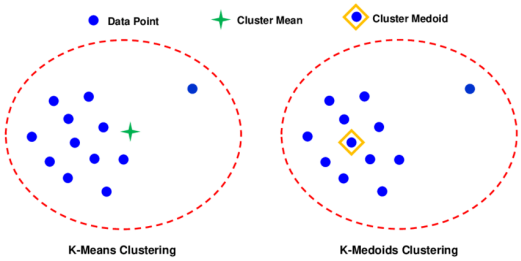

Figure 2.2: Kmeans vs Kmedoids

Source code used:

```
1  from sklearn_extra.cluster import KMedoids
2  from sklearn.metrics import silhouette_score
3
4  def kmedoid(df, nb_clusters=None, auto=False):
5      if not auto:
6          clusterer = KMedoids(n_clusters=nb_clusters)
```

```
7         labels = clusterer.fit_predict(df)
8
9         silouhette = silhouette_score(df, labels)
10
11        return {
12            'algorithm': 'kmedoid',
13            'type': 'unsupervised',
14            'silouhette': silouhette,
15            'k': nb_clusters,
16        }
17    else:
18        max_silouhette = 0
19        max_k = 0
20        for i in range(2, 30):
21            clusterer = KMedoids(n_clusters=i)
22            labels = clusterer.fit_predict(df)
23
24            silouhette = silhouette_score(df, labels)
25
26            if silouhette > max_silouhette:
27                max_silouhette = silouhette
28                max_k = i
29
30        return {
31            'algorithm': 'kmedoid',
32            'type': 'unsupervised',
33            'silouhette': float(max_silouhette),
34            'k': max_k,
35        }
```

Listing 2.3: Implémentation de K-medoids avec scikit-learn-extra

This function accepts three main parameters:

- `df` : The **pandas** dataframe.
- `nb_clusters` : The number of clusters.
- `auto` : A mode to automatically determine the best parameter k to maximize performance.

2.1.3 DBSCAN

The DBSCAN algorithm (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering method. Unlike K-means or K-

medoids, it does not require specifying a number of clusters and can identify arbitrarily shaped groups while marking isolated points as noise 2.3. The implementation is also based on **scikit-learn**.

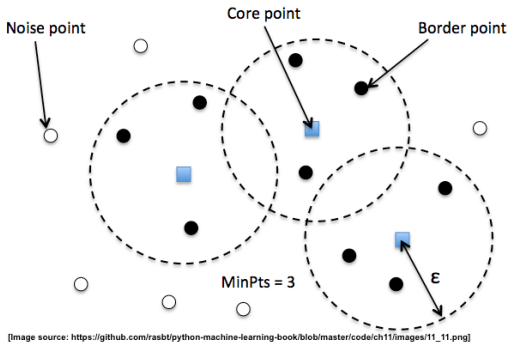

Figure 2.3: DBSCAN with outlier points

Source code used:

```
1  from sklearn.cluster import DBSCAN
2  from sklearn.metrics import silhouette_score
3
4  def dbscan(df, eps, min_samples):
5      clusterer = DBSCAN(eps=eps, min_samples=min_samples)
6      labels = clusterer.fit_predict(df)
7
8      silouhette = silhouette_score(df, labels) if len(set(labels)) >
           1 else -1
9
10     return {
11         'algorithm': 'dbscan',
12         'type': 'unsupervised',
13         'silouhette': silouhette,
14         'eps': eps,
15         'min_samples': min_samples,
16     }
```

Listing 2.4: Implémentation de DBSCAN avec scikit-learn

This function accepts three main parameters:

- `df` : The **pandas** dataframe.
- `eps` : The maximum distance between two points for them to be considered neighbors.

- `min_samples` : The minimum number of points required for a region to be considered a cluster.

2.1.4 AGNES (Agglomerative Clustering)

The AGNES algorithm (AGglomerative NESting) is a bottom-up hierarchical method. It starts by considering each point as an individual cluster, then iteratively merges clusters based on their proximity until a certain criterion is reached 2.4. The implementation is based on **scikit-learn**.

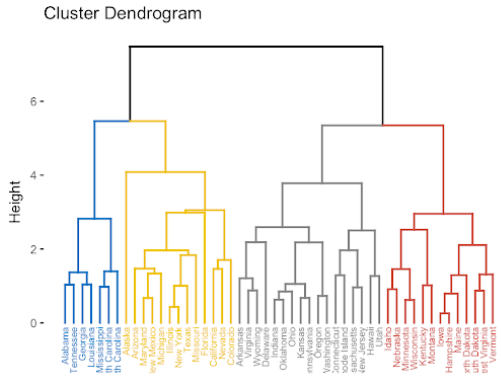

Figure 2.4: AGNES Dendrogram

Source code used:

```
1  from sklearn.cluster import AgglomerativeClustering
2  from sklearn.metrics import silhouette_score
3
4  def agnes(df, nb_clusters=None, auto=False):
5      if not auto:
6          clusterer = AgglomerativeClustering(n_clusters=nb_clusters)
7          labels = clusterer.fit_predict(df)
8
9          silouhette = silhouette_score(df, labels)
10
11         return {
12             'algorithm': 'agnes',
13             'type': 'unsupervised',
14             'silouhette': silouhette,
15             'k': nb_clusters,
16         }
17
18     else:
```

```
19          max_silouhette = 0
20          max_k = 0
21
22          for i in range(2, 30):
23              clusterer = AgglomerativeClustering(n_clusters=i)
24              labels = clusterer.fit_predict(df)
25
26              silouhette = silhouette_score(df, labels)
27
28              if silouhette > max_silouhette:
29                  max_silouhette = silouhette
30                  max_k = i
31
32          return {
33              'algorithm': 'agnes',
34              'type': 'unsupervised',
35              'silouhette': max_silouhette,
36              'k': max_k,
37          }
```

Listing 2.5: Implémentation de AGNES avec scikit-learn

This function accepts three main parameters:

- `df` : The **pandas** dataframe.
- `nb_clusters` : The desired number of clusters.
- `auto` : A mode to automatically determine the best parameter k.

2.2 Supervised Classification Algorithms

In this section, we address the supervised classification algorithms, which require labeled data to learn to predict the labels or classes of new instances.

As a performance measure, we used **accuracy**, which evaluates the overall effectiveness of a model by computing the percentage of correct predictions relative to the total number of predictions made.

Accuracy is defined by the following formula:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

This measure takes into account the proportion of instances correctly classified by the model. An accuracy of 1 (or 100%) indicates that all predictions are correct, while an accuracy close to 0 suggests poor performance.

However, it is important to note that accuracy can be misleading in the case of imbalanced data, where one class may be over-represented. For example, a model could achieve high accuracy simply by systematically predicting the majority class. In such cases, other metrics such as precision, recall, or the F_1 score may be necessary to complement the evaluation.

In our study, accuracy was used as the main measure to compare the performance of the different supervised classification algorithms on various datasets. This metric allowed us to identify the algorithms best suited to each type of analyzed data.

2.2.1 KNN (K-Nearest Neighbors)

The KNN (K-Nearest Neighbors) algorithm is a supervised algorithm used for classification. It is based on the principle of finding the k nearest neighbors of a given point in the feature space, then assigning the most frequent class among those neighbors 2.5. The value of k is a key hyperparameter of the algorithm, which can be adjusted to optimize performance. The following code implements KNN using **scikit-learn**.

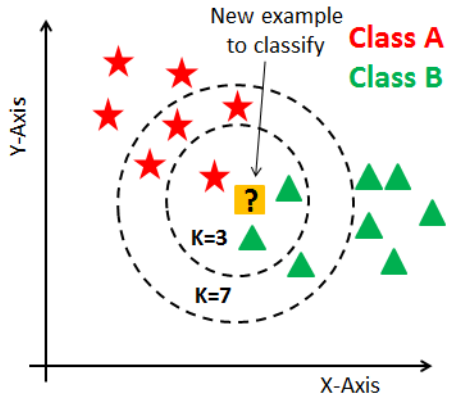

Figure 2.5: KNN

Source code used:

```
1  from sklearn.neighbors import KNeighborsClassifier
2  from sklearn.metrics import accuracy_score
3
4  def knn(df, k=None, auto=False):
5      if not auto:
6          model = KNeighborsClassifier(n_neighbors=k)
7          model.fit(df.X_train, df.y_train)
```

```
8         y_hat = model.predict(df.X_test)
9
10        accuracy = accuracy_score(df.y_test, y_hat)
11
12        return {
13            'algorithm': 'knn',
14            'type': 'supervised',
15            'k': k,
16            'accuracy': accuracy,
17        }
18
19    else:
20        max_accuracy = 0
21        max_k = 0
22
23        for k in range(1, min(30, df.df.shape[1])):
24            model = KNeighborsClassifier(n_neighbors=k)
25            model.fit(df.X_train, df.y_train)
26            y_hat = model.predict(df.X_test)
27
28            accuracy = accuracy_score(df.y_test, y_hat)
29
30            if accuracy > max_accuracy:
31                max_accuracy = accuracy
32                max_k = k
33
34        return {
35            'algorithm': 'knn',
36            'type': 'supervised',
37            'k': max_k,
38            'accuracy': max_accuracy,
39        }
```

Listing 2.6: Implémentation de KNN avec scikit-learn

This function accepts three main parameters:

- `df` : The dataframe object of the `DF` class.
- `k` : The number of neighbors (hyperparameter).
- `auto` : A mode to automatically determine the best value of k.

2.2.2 Naive Bayes

The Naive Bayes algorithm is a probabilistic classification method based on Bayes' theorem. It assumes that the features are independent of each other, hence the term "naive" 2.6.

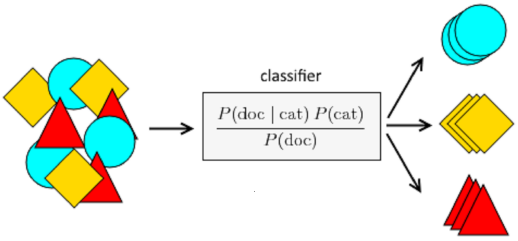

Figure 2.6: Naive Bayes

Source code used:

```
1  from sklearn.naive_bayes import GaussianNB
2  from sklearn.metrics import accuracy_score
3
4  def naive_bayes(df):
5      gnb = GaussianNB()
6      gnb.fit(df.X_train, df.y_train)
7      y_hat = gnb.predict(df.X_test)
8
9      accuracy = accuracy_score(df.y_test, y_hat)
10
11     return {
12         'algorithm': 'naive bayes',
13         'type': 'supervised',
14         'accuracy': accuracy,
15     }
```

Listing 2.7: Implémentation de Naive Bayes avec scikit-learn

This function accepts a single parameter, which is `df` the dataframe object of the `DF` class.

2.2.3 Decision Tree

The decision tree algorithm is a supervised model used for classification and regression. It divides the feature space into subspaces based on decision rules 2.7.

Source code used:

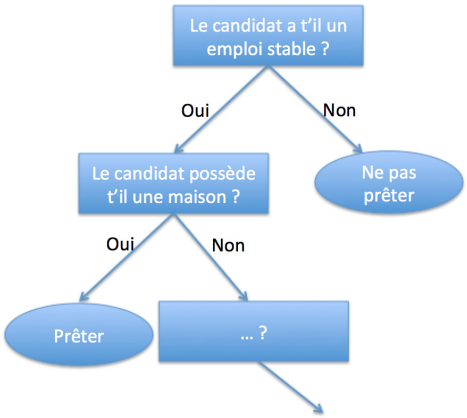

Figure 2.7: Decision Tree

```
1  from sklearn.tree import DecisionTreeClassifier
2  from sklearn.metrics import accuracy_score
3
4  def decision_tree(df):
5      clf = DecisionTreeClassifier()
6      clf.fit(df.X_train, df.y_train)
7      y_hat = clf.predict(df.X_test)
8      accuracy = accuracy_score(df.y_test, y_hat)
9
10     return {
11         'algorithm': 'decision tree',
12         'type': 'supervised',
13         'accuracy': accuracy,
14     }
```

Listing 2.8: Implémentation de Decision Tree avec scikit-learn

This function accepts a single parameter, which is `df` the dataframe object of the `DF` class.

2.2.4 Support Vector Machine (SVM)

The SVM algorithm is a supervised method used for classification. It seeks to maximize the margin between classes by projecting the data into a higher-dimensional feature space 2.8.

Source code used:

```
1  from sklearn.svm import SVC
```

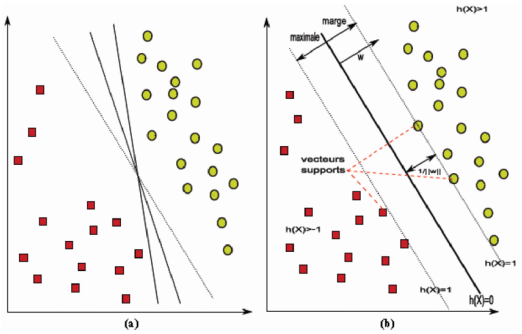

Figure 2.8: SVMs

```
from sklearn.metrics import accuracy_score
from tqdm import tqdm

def svm(df):
    model = SVC(kernel='linear')
    model.fit(df.X_train, df.y_train)
    y_hat = model.predict(df.X_test)

    accuracy = accuracy_score(df.y_test, y_hat)

    return {
        'algorithm': 'svm',
        'type': 'supervised',
        'accuracy': accuracy,
    }
```

Listing 2.9: Implémentation de SVM avec scikit-learn

This function accepts a single parameter, which is `df` the dataframe object of the `DF` class.

2.2.5 Deep Neural Networks (DNN)

Deep neural networks (DNN) are supervised models composed of multiple hidden layers of neurons. They are capable of learning complex representations of data, making them powerful for classification and regression tasks 2.9.

Source code used:

```
from keras import Sequential
from sklearn.metrics import accuracy_score
```

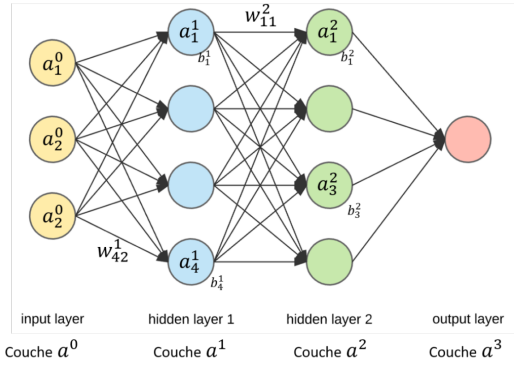

Figure 2.9: Deep Neural Network

```
3  import keras
4  import numpy as np
5
6  def dnn(df, nb_hidden_layers, nb_nodes, target_column):
7      max_accuracy = 0
8      max_nb_hidden = 0
9      max_nb_nodes = 0
10
11     for nb_hidden in nb_hidden_layers:
12         for nb_node in nb_nodes:
13             accuracy = 0
14             nb_attributs = df.X_test.shape[1]
15
16             model = Sequential()
17
18             model.add(keras.layers.Dense(nb_attributs,
                   activation='relu'))
19             for _ in range(nb_hidden):
20                     model.add(keras.layers.Dense(nb_node,
                           activation='relu'))
21
22             nb_classes = df.df[target_column].nunique()
23
24             if nb_classes == 2:
25                 model.add(keras.layers.Dense(1,
                       activation='sigmoid'))
26
27                 model.compile(optimizer='adam',
                       loss='binary_crossentropy', metrics=['accuracy'])
28
```

```
29                model.fit(df.X_train, df.y_train, epochs=50)
30                y_hat = model.predict(df.X_test)
31                y_hat = (y_hat >= 0.5).astype(int)
32
33                accuracy = accuracy_score(df.y_test, y_hat)
34
35            else:
36                model.add(keras.layers.Dense(nb_classes,
                      activation='softmax'))
37
38                model.compile(optimizer='adam',
                      loss='sparse_categorical_crossentropy',
                      metrics=['accuracy'])
39
40                model.fit(df.X_train, df.y_train, epochs=50)
41                y_hat = model.predict(df.X_test)
42                y_hat = np.argmax(y_hat, axis=1)
43
44                accuracy = accuracy_score(df.y_test, y_hat)
45
46            if accuracy > max_accuracy:
47                max_accuracy = accuracy
48                max_nb_hidden = nb_hidden
49                max_nb_nodes = nb_node
50
51    return {
52        'algorithm': 'dnn',
53        'type': 'supervised',
54        'accuracy': max_accuracy,
55        'nb hidden layers': max_nb_hidden,
56        'nb nodes per hidden layer': max_nb_nodes,
57    }
```

Listing 2.10: Implémentation de DNN avec Keras

This function accepts four main parameters:

- `df` : The dataframe object of the `DF` class.
- `nb_hidden_layers` : A list of one or more values corresponding to the number of hidden layers.
- `nb_nodes` : A list of one or more values corresponding to the number of nodes in a layer.

- `target_column` : The name of the class attribute of the dataset.

Chapter 3

Hyperparameter Optimization for Each Algorithm

In this section, we will present the results obtained for the optimization of each algorithm on each dataset.

3.1 Unsupervised Classification

For algorithms such as **kmeans**, **kmedoids** and **AGNES**, we varied the number of clusters k. And for **DBSCAN** we varied *epsilon* and *min pts*.

	Kmeans		Kmedoid		Agnes		DBscan		
Hyperparameters	Nb of clusters	Score max	Nb of clusters	Score max	Nb of clusters	Score max	epsilon	min points	Score max
breast.csv	2	0.35	2	0.34	2	0.33	1.5	10	0.06
contact-lenses.arff	3	0.24	2	0.08	3	0.24	1.5	2	0.23
diabetes.arff	2	0.57	2	0.15	6	0.16	1.5	20	0.11
ecoli.csv	2	0.82	3	0.42	5	0.46	1.5	3	0.33
heart.csv	2	0.18	3	0.08	2	0.17	1.5	20	0.11
hepatitis.csv	2	0.22	2	0.12	2	0.25	2.5	5	0.01
hepatitisequilibre.arff	2	0.78	2	0.2	2	0.21	2	5	0.03
horse-colic.csv	2	0.99	2	0.1	9	0.11	4	5	0.07
hypothyroid.arff	2	0.67	5	0.03	9	0.3	0.5	20	0.26
IRIS 1.csv	2	0.59	2	0.59	2	0.59	1.5	2	0.59
lymph.arff	2	0.43	2	0.1	2	0.16	3	5	0.26

Table 3.1: Table summarizing the results found

The results can be summarized as follows for better readability:
We notice that **Kmeans** performed very well on most datasets.

3.2 Supervised Classification

For the **KNN** algorithm, we varied k. And for the **Deep Neural Network** we varied *the number of hidden layers* and *the number of nodes per hidden layer*.

Dataset	**Best Algorithm**
breast.csv	Kmeans
contact-lenses.arff	Kmeans and AGNES
diabetes.arff	Kmeans
ecoli.csv	Kmeans
heart.csv	Kmeans
hepatitis.csv	AGNES
hepatitisequilibre.arff	Kmeans
horse-colic.csv	Kmeans
hypothyroid.arff	Kmeans
IRIS 1.csv	Kmeans, Kmedoid, AGNES and DBSCAN
lymph.arff	Kmeans

Table 3.2: Best algorithm for each dataset

We obtain the following results:

	KNN		**Naive Bayes**	**Decision Tree**	**SVM**	**DNN**		
Hyperparameters	Number of neighbors	**Accuracy**	**Accuracy**	**Accuracy**	**Accuracy**	Nb hidden layers	Nb nodes	**Accuracy**
breast.csv	3	98%	93%	98%	95%	6	8	96%
contact-lenses.arff	4	60.00%	40.00%	60%	60%	6	8	100%
diabetes.arff	1	68%	73.38%	72.08%	74.02%	2	8	77.27%
ecoli.csv	2	80.59%	70.14%	77.61%	80.59%	6	16	52.23%
heart.csv	4	82.75%	77.58%	75.86%	86.20%	2	4	93.10%
hepatitis.csv	9	68.96%	65.51%	72.41%	68.93%	6	16	62.06%
hepatitisequilibre.arff	2	92.85%	78.57%	92.85%	85.71%	2	8	92.86%
horse-colic.csv	2	71.66%	60%	86.66%	66.66%	4	16	80%
hypothyroid.arff	5	93.37%	18.67%	99.73%	97.08%	2	4	92.84%
IRIS 1.csv	6	93.33%	90%	90%	90%	2	4	60%
lymph.arff	4	90%	86.66%	90%	80%	2	4	43.33%

Table 3.3: Table summarizing the results of supervised classification

The results can be summarized as follows for better readability:

For supervised classification, we notice that **KNN** and **Decision Tree** perform very well.

Dataset	**Best Algorithm**
breast.csv	KNN and Decision Tree
contact-lenses.arff	DNN
diabetes.arff	DNN
ecoli.csv	KNN and SVM
heart.csv	DNN
hepatitis.csv	Decision Tree
hepatitisequilibre.arff	KNN, Decision Tree and DNN
horse-colic.csv	Decision Tree
hypothyroid.arff	Decision Tree
IRIS 1.csv	KNN
lymph.arff	KNN and Decision Tree

Table 3.4: Best supervised classification algorithm for each dataset

Conclusion

After conducting several tests on the previously mentioned datasets, we can conclude that the best performing unsupervised classification algorithm across all datasets is **K-means**. Regarding supervised classification, we found that both **KNN** and **Decision Tree** algorithms perform very well.