République Algérienne Démocratique et Populaire
Université des Sciences et de la Technologie HOUARI BOUMEDIENE
Faculté d'Informatique

Rapport Data Mining Sur Les Algorithmes Supervisés et Non Supervisés

Rédigé par
— Mouhoub Massinissa

Table des matières

Introduction

Le *data mining*, ou exploration de données, est une discipline essentielle dans le domaine de la science des données. Elle consiste à extraire des connaissances utiles à partir de grands ensembles de données, souvent complexes et hétérogènes. L'objectif principal du *data mining* est de découvrir des modèles, des relations ou des tendances qui peuvent guider les prises de décision dans divers domaines, notamment la santé, la finance, le marketing et bien d'autres secteurs.

Dans le cadre de ce rapport, nous explorons deux approches fondamentales en *data mining* : les algorithmes supervisés et les algorithmes non supervisés. Ces deux types d'algorithmes jouent un rôle crucial dans le traitement et l'analyse des données, chacun ayant des objectifs et des méthodes distincts. Les algorithmes supervisés reposent sur des ensembles de données labellisées, permettant de prédire des étiquettes ou des valeurs pour de nouvelles données. Les algorithmes non supervisés, quant à eux, travaillent sur des données non labellisées et se concentrent sur l'identification de structures ou de regroupements inhérents.

Ce rapport vise à présenter une analyse complète de plusieurs algorithmes supervisés et non supervisés, en mettant l'accent sur leurs définitions, leurs principes de fonctionnement et leurs applications pratiques. Une partie essentielle de cette étude consiste à évaluer les performances de ces algorithmes sur différents ensembles de données, à optimiser leurs hyperparamètres et à déterminer les scénarios les plus adaptés à leur utilisation.

En complément, nous présentons une interface utilisateur intuitive permettant de manipuler les algorithmes et de visualiser leurs résultats de manière interactive. Cette interface vise à simplifier l'expérimentation et à rendre les concepts abordés plus accessibles et adaptés pour tous.

Ainsi, ce rapport se structure autour des éléments suivants :

1. Définitions théoriques.
2. Code source utilisé.
3. Optimisation des hyperparamètres pour chaque algorithme.

Chapitre 1

Définition Théorique des Algorithmes de Classification

1.1 Définition de la Classification Non Supervisée

La classification non supervisée, ou clustering, est une méthode d'apprentissage automatique où l'on cherche à regrouper des données sans étiquettes préalables en ensembles homogènes. L'objectif est de découvrir des structures cachées ou des motifs intrinsèques dans les données, en identifiant des groupes ou des clusters de points similaires.

1.2 Définition de la Classification Supervisée

La classification supervisée est une méthode d'apprentissage automatique où un modèle est entraîné sur un ensemble de données étiquetées pour prédire les étiquettes de nouvelles données. Le modèle apprend la relation entre les caractéristiques d'entrée et les étiquettes de sortie, permettant des prédictions sur des données inconnues.

1.3 Comparaison entre Classification Non Supervisée et Classification Supervisée

La comparaison entre les deux approches peut être résumée dans le tableau suivant :

Aspect	**Classification Non Supervisée**	**Classification Super**
Données requises	Non labellisées	Labellisées
Objectif	Découverte de structures	Prédiction de label
Applications typiques	Exploration de données, segmentation	Reconnaissance, classification
Exemples d'algorithmes	K-means, DBSCAN	KNN, Decision Trees,

TABLE 1.1 – Comparaison entre classification non supervisée et supervisée.

1.3.1 Choix de l'Approche

Le choix entre une approche supervisée ou non supervisée dépend fortement du problème à résoudre et de la disponibilité des données :

— Si des labels sont disponibles, la classification supervisée est généralement la meilleure option pour des tâches prédictives.

— En l'absence de labels, la classification non supervisée peut être utile pour explorer les données ou identifier des patterns cachés.

Chapitre 2

Code Source Utilisé

Dans cette section, nous allons expliquer les éléments clés du code utilisé pour obtenir les résultats présentés. Cette explication servira également de guide pour mieux comprendre le fonctionnement du logiciel.

Nous avons utilisé des fonctions déjà implémentées dans des bibliothèques largement reconnues et adoptées sur le marché, telles que **scikit-learn** et **TensorFlow**. Ces bibliothèques bénéficient d'une excellente réputation et d'une très grande communauté de développeurs, ce qui garantit leur fiabilité et leur robustesse.

Nous commencerons par présenter les fonctions utilisées pour exécuter les algorithmes de classification.

2.1 Algorithmes de Classification Non Supervisée

Dans cette section, nous abordons les algorithmes de classification non supervisée, qui permettent d'identifier des regroupements dans les données sans nécessiter d'étiquettes préalables.

Comme mesure de performance, nous avons travaillé avec le score silhouette, qui permet de mesurer la qualité du regroupement en évaluant à quel point les points de données sont bien assignés à leurs clusters respectifs tout en étant distincts des autres clusters.

Le score silhouette est calculé pour chaque point de données en prenant en compte deux valeurs principales :

- **La cohésion intra-cluster** (a) : la distance moyenne entre un point et les autres points de son propre cluster.
- **La séparation inter-cluster** (b) : la distance moyenne entre un point et les points du cluster le plus proche auquel il n'appartient pas.

La formule du score silhouette pour un point est donnée par :

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

où s varie de -1 à 1. Un score proche de 1 indique que le point est bien assigné à son cluster, tandis qu'un score proche de -1 suggère une mauvaise assignation.

Dans notre étude, le score silhouette a été utilisé pour évaluer et comparer les performances des différents algorithmes de classification non supervisée sur divers ensembles de données.

Afin de faciliter la gestion du dataframe **pandas**, on a créé une classe qui permet d'automatiser plein de choses :

```
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

L'algorithme K-means est un algorithme de partitionnement des données qui divise les points en K clusters en minimisant la variance intra-cluster. L'implémentation que nous avons utilisée repose sur la bibliothèque **scikit-learn**, qui propose une méthode optimisée et facile à utiliser pour le clustering 2.1. Le processus principal est le suivant :

1. Initialisation des K centroïdes de manière aléatoire ou en utilisant une méthode comme `k-means++`.
2. Assignation de chaque point de données au centroïde le plus proche.
3. Mise à jour des positions des centroïdes en calculant la moyenne des points attribués à chaque cluster.
4. Répétition des étapes 2 et 3 jusqu'à convergence (c'est-à-dire lorsque les centroïdes ne changent plus significativement ou qu'un nombre maximal d'itérations est atteint).

Code source utilisé :

```
1 from sklearn.cluster import KMeans
2 from sklearn.metrics import silhouette_score
3
4 def kmeans(df, nb_clusters=None, auto=False):
```

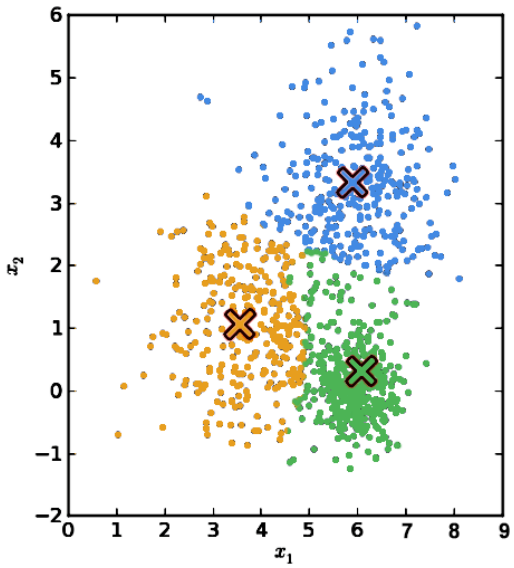

FIGURE 2.1 – Kmeans

```
5       if not auto:
6           clusterer = KMeans(n_clusters=nb_clusters)
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

Cette fonction accepte 3 attributs :
— `df` C'est le dataframe **pandas**.
— `nb_clusters` Correspond au nombre de clusters.
— `auto` Permet d'automatiser l'algorithme afin d'avoir le meilleur parametere pour maximiser la performance..

2.1.2 K-medoids

L'algorithme K-medoids est une variante robuste de K-means qui utilise des points réels comme centroïdes, réduisant ainsi l'impact des valeurs aberrantes. L'implémentation repose sur la bibliothèque **scikit-learn-extra**, offrant une méthode efficace pour le partitionnement. Le processus principal est similaire à K-means 2.2, mais les centroïdes sont choisis parmi les points existants.

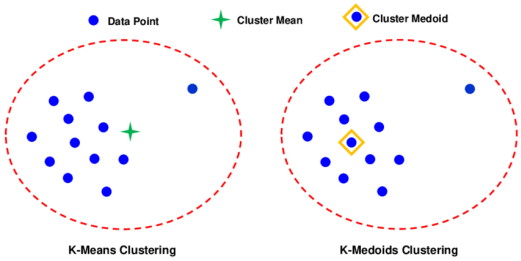

FIGURE 2.2 – Kmeans vs Kmedoids

Code source utilisé :

```
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

Cette fonction accepte trois paramètres principaux :
— `df` : Le dataframe **pandas**.
— `nb_clusters` : Le nombre de clusters.
— `auto` : Un mode pour déterminer automatiquement le meilleur paramètre k pour maximiser la performance.

2.1.3 DBSCAN

L'algorithme DBSCAN (Density-Based Spatial Clustering of Applications with Noise) est une méthode de clustering basée sur la densité. Contrairement

à K-means ou K-medoids, il ne nécessite pas de spécifier un nombre de clusters et peut identifier des groupes de forme arbitraire tout en marquant les points isolés comme du bruit 2.3. L'implémentation repose également sur **scikit-learn**.

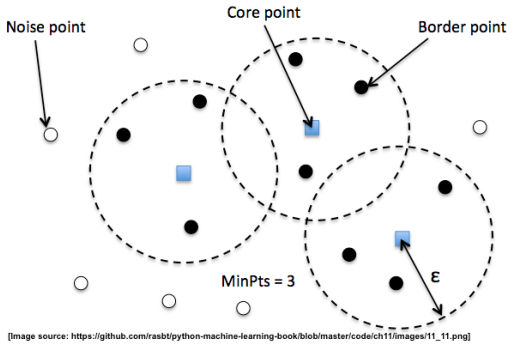

FIGURE 2.3 – DBSCAN avec des points outliers

Code source utilisé :

```python
from sklearn.cluster import DBSCAN
from sklearn.metrics import silhouette_score

def dbscan(df, eps, min_samples):
    clusterer = DBSCAN(eps=eps, min_samples=min_samples)
    labels = clusterer.fit_predict(df)

    silouhette = silhouette_score(df, labels) if len(set(labels)) > 1 else -1

    return {
        'algorithm': 'dbscan',
        'type': 'unsupervised',
        'silouhette': silouhette,
        'eps': eps,
        'min_samples': min_samples,
    }
```

Listing 2.4 – Implémentation de DBSCAN avec scikit-learn

Cette fonction accepte trois paramètres principaux :
— `df` : Le dataframe **pandas**.
— `eps` : La distance maximale entre deux points pour qu'ils soient considérés comme voisins.

— `min_samples` : Le nombre minimum de points nécessaires pour qu'une région soit considérée comme un cluster.

2.1.4 AGNES (Agglomerative Clustering)

L'algorithme AGNES (AGglomerative NESting) est une méthode hiérarchique ascendante. Il commence par considérer chaque point comme un cluster individuel, puis fusionne les clusters de manière itérative en fonction de leur proximité jusqu'à ce qu'un certain critère soit atteint 2.4. L'implémentation repose sur **scikit-learn**.

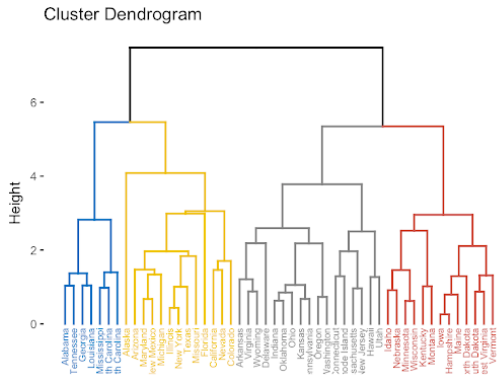

FIGURE 2.4 – Dendrogramme de AGNES

Code source utilisé :

```
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import silhouette_score

def agnes(df, nb_clusters=None, auto=False):
    if not auto:
        clusterer = AgglomerativeClustering(n_clusters=nb_clusters)
        labels = clusterer.fit_predict(df)

        silouhette = silhouette_score(df, labels)

        return {
            'algorithm': 'agnes',
            'type': 'unsupervised',
            'silouhette': silouhette,
            'k': nb_clusters,
        }

```

```
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

Cette fonction accepte trois paramètres principaux :
— `df` : Le dataframe **pandas**.
— `nb_clusters` : Le nombre de clusters souhaité.
— `auto` : Un mode pour déterminer automatiquement le meilleur paramètre k.

2.2 Algorithmes de Classification Supervisée

Dans cette section, nous abordons les algorithmes de classification supervisée, qui nécessitent des données labellisées pour apprendre à prédire les étiquettes ou classes des nouvelles instances.

Comme mesure de performance, nous avons utilisé l'**accuracy**, qui permet d'évaluer l'efficacité globale d'un modèle en calculant le pourcentage de prédictions correctes par rapport au nombre total de prédictions réalisées.

L'accuracy est définie par la formule suivante :

$$\text{Accuracy} = \frac{\text{Nombre de prédictions correctes}}{\text{Nombre total de prédictions}}$$

Cette mesure prend en compte la proportion d'instances correctement classées par le modèle. Une accuracy de 1 (ou 100%) indique que toutes les

prédictions sont correctes, tandis qu'une accuracy proche de 0 suggère une performance médiocre.

Cependant, il est important de noter que l'accuracy peut être trompeuse dans le cas de données déséquilibrées, où une classe peut être sur-représentée. Par exemple, un modèle pourrait obtenir une accuracy élevée simplement en prédisant systématiquement la classe majoritaire. Dans ces cas, d'autres métriques comme la précision, le rappel ou le score F_1 peuvent être nécessaires pour compléter l'évaluation.

Dans notre étude, l'accuracy a été utilisée comme mesure principale pour comparer les performances des différents algorithmes de classification supervisée sur divers ensembles de données. Cette métrique a permis d'identifier les algorithmes les mieux adaptés à chaque type de données analysées.

2.2.1 KNN (K-Nearest Neighbors)

L'algorithme KNN (K-Nearest Neighbors) est un algorithme supervisé utilisé pour la classification. Il repose sur le principe de trouver les k voisins les plus proches d'un point donné dans l'espace des caractéristiques, puis d'affecter la classe la plus fréquente parmi ces voisins 2.5. La valeur de k est un hyperparamètre clé de l'algorithme, qui peut être ajusté pour optimiser la performance. Le code suivant implémente KNN en utilisant **scikit-learn**.

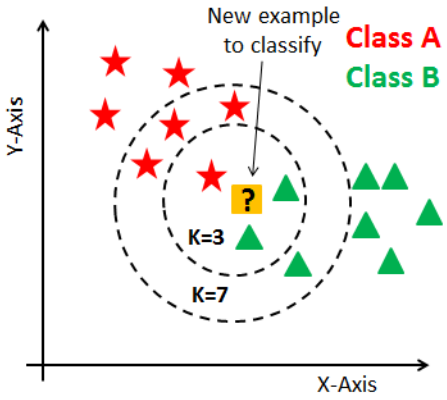

FIGURE 2.5 – KNN

Code source utilisé :

```
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

Cette fonction accepte trois paramètres principaux :
— `df` : L'objet dataframe de la class `DF`.
— `k` : Le nombre de voisins (hyperparamètre).
— `auto` : Un mode pour déterminer automatiquement la meilleure valeur de k.

2.2.2 Naive Bayes

L'algorithme Naive Bayes est une méthode de classification probabiliste basée sur le théorème de Bayes. Il suppose que les caractéristiques sont indépendantes les unes des autres, d'où le terme "naive" 2.6.

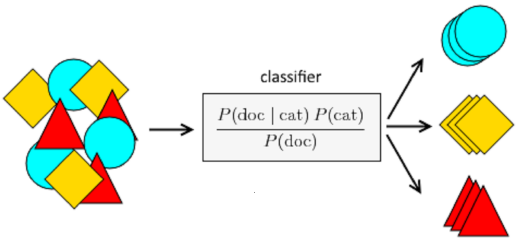

FIGURE 2.6 – Naive Bayes

Code source utilisé :

```
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

Cette fonction accepte un seul paramètre, qui est `df` l'objet dataframe de la classe `DF`.

2.2.3 Arbre de Décision (Decision Tree)

L'algorithme d'arbre de décision est un modèle supervisé utilisé pour la classification et la régression. Il divise l'espace des caractéristiques en sous-espaces basés sur des règles de décision 2.7.

Code source utilisé :

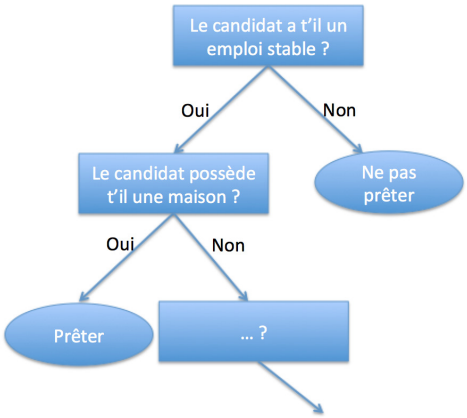

FIGURE 2.7 – Decision Tree

```
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

Cette fonction accepte un seul paramètre, qui est `df` l'objet dataframe de la classe `DF`.

2.2.4 Support Vector Machine (SVM)

L'algorithme SVM est une méthode supervisée utilisée pour la classification. Il cherche à maximiser la marge entre les classes en projetant les données dans un espace de caractéristiques de plus grande dimension 2.8.

Code source utilisé :

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

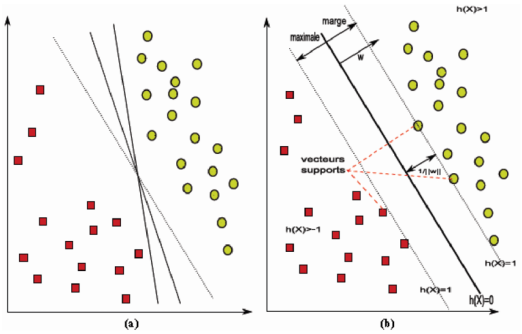

FIGURE 2.8 – SVMs

```
2  from sklearn.metrics import accuracy_score
3  from tqdm import tqdm
4
5  def svm(df):
6      model = SVC(kernel='linear')
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': 'svm',
14         'type': 'supervised',
15         'accuracy': accuracy,
16     }
```

Listing 2.9 – Implémentation de SVM avec scikit-learn

Cette fonction accepte un seul paramètre, qui est `df` l'objet dataframe de la classe `DF`.

2.2.5 Deep Neural Networks (DNN)

Les réseaux neuronaux profonds (DNN) sont des modèles supervisés composés de plusieurs couches cachées de neurones. Ils sont capables d'apprendre des représentations complexes des données, ce qui les rend puissants pour des tâches de classification et de régression 2.9.

Code source utilisé :

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

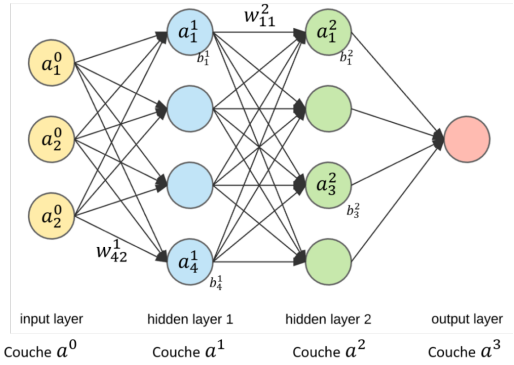

FIGURE 2.9 – Deep Neural Network

```
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

Cette fonction accepte quatre paramètres principaux :
— `df` : L'objet dataframe de la classe `DF`.
— `nb_hidden_layers` : Une liste de une ou plusieurs valeurs qui correspond au nombre de couche cachées.
— `nb_nodes` : Une liste de une ou plusieurs valeurs qui correspond au nombre noeuds dans une couche.
— `target_column` : Le nom de l'attribut classe du dataset.

Chapitre 3

Optimisation des hyperparamètres pour chaque algorithme

Dans cette section, nous présenterons les résultats obtenus pour l'optimisation de chaque algorithme sur chaque dataset

3.1 Classification Non Supervíse

Pour les algorithmes tels que **kmeans**, **kmedoids** et **AGNES**, nous avons fait varier le nombre de clusters *k*. Et pour **DBSCAN** nous avons varié *epsilon* et *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 – Tableau résumant les résultats trouvés

On peut résumer les résultats comme ci-dessous afin de mieux voir :

On remarque que **Kmeans** a été très performant sur la plupart des datasets.

Dataset	Meilleur Algorithme
breast.csv	Kmeans
contact-lenses.arff	Kmeans et 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 et DBSCAN
lymph.arff	Kmeans

TABLE 3.2 – Meilleur algorithme pour chaque dataset

3.2 Classification Supervisé

Pour l'algorithme **KNN**, on a fait varier le *k*. Et pour le **Deep Neural Network** on a fait varier *le nombre de couche cachées* et *le nombre noeuds pour chaque couche cachée*.

On obtient les résultats suivants :

	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 – Tableau qui résume les résultats de la classification supervisé

On peut résumer les résultats comme ci-dessous afin de mieux voir :

Pour la classification supervisée, on remarque que le **KNN** et le **Decision Tree** performent très bien.

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

TABLE 3.4 – Meilleur algorithme de classification supervisé pour chaque dataset

Conclusion

Après avoir effectué plusieurs tests sur les datasets mentionnés précédemment, nous pouvons conclure que l'algorithme de classification non supervisée le plus performant pour l'ensemble des datasets est **K-means**. En ce qui concerne la classification supervisée, nous avons constaté que les deux algorithmes **KNN** et **Decision Tree** performent très bien.