Machine Learning
Projet
M1 MIASHS

Nom : Tiemtoré Wendyam Ariel

Contexte du projet

Ce projet s'inscrit dans le cadre de l'apprentissage supervisé appliqué à la classification binaire. L'objectif est d'explorer différentes approches de classification afin d'analyser leur performance sur des jeux de données spécifiques.

Objectifs

1. Préparer et analyser les données
2. Implémenter et comparer différentes méthodes de classification
3. Évaluer les performances des modèles
4. Comparer les temps d'apprentissage et d'exécution

Préparation des Données

Importation et exploration des données

Nous avons utilisé le jeu de données bankmarketing.csv. La dernière colonne représente l'étiquette à prédire.

Prétraitement des données

Pour le jeu de donnée choisi, nous avons vérifié les données et nous n'avons pas rencontré de valeurs manquantes. Les variables que nous avons dans la base sont des variables discrètes, ce qui nous épargne un travail de normalisation des variables continues. Cependant dans la suite de notre travail nous avons séparé les données en ensemble d'entraînement (70%) et de test (30%).

Présentation des données

Le script commence par charger et explorer les données. Voici ce que nous observons :

Structure des données :

Le jeu de données contient 45 211 entrées (lignes) et 52 colonnes.

Les colonnes sont nommées de X1 à X51, et la dernière colonne est y, qui représente la variable cible (étiquette à prédire).

Toutes les colonnes sont de type float64, ce qui suggère que les variables catégorielles ont déjà été encodées numériquement.

Aperçu des données :

Les 5 premières lignes du jeu de données sont affichées. On remarque que la plupart des valeurs sont soit 0.0, soit 1.0, ce qui indique que les variables catégorielles ont été encodées en one-hot encoding ou binaire.

La colonne y (variable cible) contient des valeurs 0.0 ou 1.0, ce qui confirme qu'il s'agit d'un problème de classification binaire.

Distribution de la variable cible :

La variable cible y est déséquilibrée : 0.0 apparaît 39 922 fois tandis que 1.0 apparaît 5 289 fois. Cela signifie que la classe majoritaire (0) est beaucoup plus fréquente que la classe minoritaire (1).

Méthodes et Algorithmes Utilisés

Le script teste plusieurs algorithmes de classification binaire, conformément aux instructions du projet. Voici les méthodes utilisées :

a. Approches non paramétriques

k-NN (k-Nearest Neighbors) : Le modèle k-NN est entraîné avec k=5. Accuracy : **87,92 %.**

Condensed Nearest Neighbor (CNN) : Une variante de k-NN qui réduit l'ensemble d'entraînement en conservant uniquement les points nécessaires. Accuracy : **87,92 %** (identique à k-NN dans ce cas).

b. Approches paramétriques linéaires

SVM linéaire : Un classifieur linéaire utilisant un noyau linéaire. Accuracy : **88,77 %.**

SVM avec noyau gaussien : Un SVM utilisant un noyau gaussien (RBF) pour capturer des relations non linéaires. Accuracy : **88,22 %.**

Régression logistique : Un modèle linéaire pour la classification binaire. Accuracy : **89,99 %.**

c. Approches non linéaires

Arbre de décision : Un modèle basé sur des règles de décision. Accuracy : **87,55 %.**

Forêt aléatoire : Un ensemble d'arbres de décision pour améliorer la précision. Accuracy : **90,65 %** (meilleure performance parmi tous les modèles).

Adaboost : Un modèle de boosting qui combine plusieurs classifieurs faibles. Accuracy : **89,72 %.**

En résumé, on a le tableau suivant :

Modèle	**Accuracy**
k-NN	0.879 (87.9%)
Condensed NN (CNN)	0.879 (87.9%)
SVM Linéaire	0.888 (88.8%)
Régression Logistique	0.900 (90.0%)
SVM Gaussien	0.882 (88.2%)
Arbre de Décision	0.875 (87.5%)
Forêt Aléatoire	0.907 (90.7%)
AdaBoost	0.897 (89.7%)

Interprétation des résultats :

Forêt Aléatoire (90.7%) donne la **meilleure performance** parmi toutes les méthodes**,** très efficace sur des données **non linéaires** et complexes et capture bien les interactions entre les variables.

Régression Logistique (90.0%) donne un bon résultat, **solide sur des problèmes linéaires** et **facile à interpréter**.

SVM Linéaire (88.8%) vs SVM Gaussien (88.2%) : SVM **linéaire** fait un peu mieux, **les données semblent globalement bien séparables linéairement** comparativement à SVM **gaussien** légèrement inférieur, suggérant que **les frontières de décision ne nécessitent pas une transformation non linéaire trop complexe**.

k-NN (87.9%) vs CNN (87.9%) : Même performance. On remarque aussi que **CNN a bien compressé les données sans perte d'information** et k-NN est sensible à la **malédiction de la dimensionnalité**, ce qui peut expliquer pourquoi il n'est pas aussi performant.

Arbre de Décision (87.5%) est légèrement inférieur, est aussi **sensible au bruit** et à **l'overfitting**.

AdaBoost (89.7%) : Approche boosting performante, mais légèrement inférieure à la **Forêt Aléatoire**.

Analyse des résultats

- **Performances** :
    - La **forêt aléatoire** obtient la meilleure accuracy (90,65 %), suivie de près par la **régression logistique** (89,99 %) et **Adaboost** (89,72 %).

- Les modèles linéaires (SVM linéaire et régression logistique) performent bien, ce qui suggère que les données pourraient être linéairement séparables ou presque.
- Les modèles non paramétriques (k-NN et CNN) ont des performances légèrement inférieures (87,92 %), ce qui peut être dû à leur sensibilité au bruit ou à la taille du jeu de données.

On peut résumer les performances comme suit :

Approche	Modèle	Accuracy (%)	Observations
Non Paramétrique	k-NN	88.03%	Sensible à la dimensionnalité
	CNN	88.03%	Réduction de la taille sans perte de précision
Paramétrique Linéaire	SVM Linéaire	88.77%	Indique que les données sont majoritairement linéaires
	Régression Logistique	89.99%	Très bonne performance pour un modèle linéaire
Non Linéaire	SVM Gaussien	88.22%	Pas d'amélioration significative
	Arbre de Décision	87.55%	Moins performant, risque d'overfitting
	Forêt Aléatoire	90.65%	Meilleur modèle, capture bien les relations complexes

- **Temps d'apprentissage**

Le temps d'apprentissage du k-NN est de 9.5367431640625e-07 seconds contre 1.1920928955078125e-06 seconds pour le CNN. On voit alors que le k-NN est beaucoup plus rapide que le CNN.

- **Déséquilibre des classes :**

La classe majoritaire (0) est beaucoup plus fréquente que la classe minoritaire (1). Cela peut biaiser les résultats, car les modèles peuvent privilégier la prédiction de la classe majoritaire. Des techniques de rééchantillonnage (oversampling, undersampling) ou des méthodes cost-sensitive pourraient être utilisées pour améliorer les performances sur la classe minoritaire.

Conclusion

- **Meilleur modèle** : La forêt aléatoire obtient la meilleure accuracy (90,65 %), ce qui en fait le modèle le plus performant pour ce jeu de données.
- **Perspectives** :

Pour améliorer les résultats, on pourrait :

- Traiter le déséquilibre des classes en utilisant des techniques de rééchantillonnage.
- Optimiser les hyperparamètres des modèles (par exemple, augmenter max_iter pour la régression logistique).

- Explorer d'autres métriques (précision, rappel, F1-score) pour mieux évaluer les performances sur la classe minoritaire.
- Tester des modèles plus avancés (**Deep Learning**).
- Explorer d'autres stratégies d'optimisation (**Bayesian Optimization**).
- Étudier l'impact de techniques d'échantillonnage pour données déséquilibrées

Annexe

```
import pandas as pd
import time
from sklearn.preprocessing import LabelEncoder
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


# Fonction de mesure de la durée
def measure_time(func):
    start_time = time.time()
    result = func
    end_time = time.time()
    execution_time = end_time - start_time
    print(f"Execution time: {execution_time} seconds")
    return result

import pandas as pd

# Charger les données
data = pd.read_csv("C:/Users/HP/Desktop/MIASHS/TD_Guillaume/Evaluation/bankmarketing.csv")

# Afficher les premières lignes
print(data.head())

#%%
# Informations générales
print(data.info())

# Distribution de la variable cible
print(data["y"].value_counts())

#%%
```

```
from sklearn.preprocessing import LabelEncoder

# Encodage de la variable cible
label_encoder = LabelEncoder()
data["y"] = label_encoder.fit_transform(data["y"])

# Séparation des features et de la cible
X = data.drop("y", axis=1)
y = data["y"]

# Encodage des variables catégorielles (si nécessaire)
X = pd.get_dummies(X, drop_first=True)

#%%
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Séparation des données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# k-NN avec k = 5
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Prédiction et évaluation
y_pred = knn.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy k-NN: {accuracy}")
measure_time(accuracy)


from sklearn.neighbors import NearestNeighbors

# Réduction de l'ensemble d'entraînement avec CNN
cnn = NearestNeighbors(n_neighbors=1)
cnn.fit(X_train)
_, indices = cnn.kneighbors(X_train)

# Sélection des points conservés
X_train_cnn = X_train.iloc[indices.flatten()]
y_train_cnn = y_train.iloc[indices.flatten()]

# k-NN sur l'ensemble condensé
knn_cnn = KNeighborsClassifier(n_neighbors=5)
knn_cnn.fit(X_train_cnn, y_train_cnn)
```

```
# Prédiction et évaluation
y_pred_cnn = knn_cnn.predict(X_test)
accuracy_cnn = accuracy_score(y_test, y_pred_cnn)
print(f"Accuracy CNN: {accuracy_cnn}")
measure_time(accuracy_cnn)

#%%
from sklearn.svm import SVC

# SVM linéaire
svm = SVC(kernel="linear")
svm.fit(X_train, y_train)

# Prédiction et évaluation
y_pred_svm = svm.predict(X_test)
accuracy_svm = accuracy_score(y_test, y_pred_svm)
print(f"Accuracy SVM linéaire: {accuracy_svm}")
measure_time(accuracy_svm)

from sklearn.linear_model import LogisticRegression

# Régression logistique
logit = LogisticRegression(max_iter=1000)
logit.fit(X_train, y_train)

# Prédiction et évaluation
y_pred_logit = logit.predict(X_test)
accuracy_logit = accuracy_score(y_test, y_pred_logit)
print(f"Accuracy Régression logistique: {accuracy_logit}")
measure_time(accuracy_logit)

# SVM avec noyau gaussien
svm_gaussian = SVC(kernel="rbf")
svm_gaussian.fit(X_train, y_train)

# Prédiction et évaluation
y_pred_gaussian = svm_gaussian.predict(X_test)
accuracy_gaussian = accuracy_score(y_test, y_pred_gaussian)
print(f"Accuracy SVM gaussien: {accuracy_gaussian}")
measure_time(accuracy_gaussian)

#%%
from sklearn.tree import DecisionTreeClassifier

# Arbre de décision
tree = DecisionTreeClassifier()
tree.fit(X_train, y_train)
```

```
# Prédiction et évaluation
y_pred_tree = tree.predict(X_test)
accuracy_tree = accuracy_score(y_test, y_pred_tree)
print(f"Accuracy Arbre de décision: {accuracy_tree}")
measure_time(accuracy_tree)

from sklearn.ensemble import RandomForestClassifier

# Forêt aléatoire
rf = RandomForestClassifier()
rf.fit(X_train, y_train)

# Prédiction et évaluation
y_pred_rf = rf.predict(X_test)
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print(f"Accuracy Forêt aléatoire: {accuracy_rf}")
measure_time(accuracy_rf)

from sklearn.ensemble import AdaBoostClassifier

# Adaboost
adaboost = AdaBoostClassifier()
adaboost.fit(X_train, y_train)

# Prédiction et évaluation
y_pred_adaboost = adaboost.predict(X_test)
accuracy_adaboost = accuracy_score(y_test, y_pred_adaboost)
print(f"Accuracy Adaboost: {accuracy_adaboost}")
measure_time(accuracy_adaboost)
```