Conception d'un systeme d'intervention pour les eleves

March 17, 2019

1 Introduction

Le but de ce projet est d'identifier les élèves qui pourraient avoir besoin d'une intervention précoce avant l'obtention de leur diplôme.

2 Exploration des données

La cellule de code ci-dessous est pour charger les bibliothèques Python nécessaires et charger les données élèves. La dernière colonne de cet ensemble de données, " passed ", sera notre étiquette cible (que l'étudiant ait obtenu ou non son diplôme). Toutes les autres colonnes sont des caractéristiques de chaque élève.

```
In [1]: import pandas as pd
        import numpy as np
        from time import time
        from sklearn.metrics import f1_score


        student_data = pd.read_csv(r"C:\Users\acer\Desktop\student-data.csv")
        student_data.head()
```

Out[1]:

	school	sex	age	address	famsize	Pstatus	Medu	Fedu	Mjob	Fjob	...	internet	romantic	famrel	freetime	goout	Dalc	Walc	health	absences	passed
0	GP	F	18	U	GT3	A	4	4	at_home	teacher	...	no	no	4	3	4	1	1	3	6	
1	GP	F	17	U	GT3	T	1	1	at_home	other	...	yes	no	5	3	3	1	1	3	4	
2	GP	F	15	U	LE3	T	1	1	at_home	other	...	yes	no	4	3	2	2	3	3	10	
3	GP	F	15	U	GT3	T	4	2	health	services	...	yes	yes	3	2	2	1	1	5	2	
4	GP	F	16	U	GT3	T	3	3	other	other	...	no	no	4	3	2	1	2	5	4	

```
0     no
1     no
2    yes
3    yes
4    yes

[5 rows x 31 columns]
```

```
In [2]: # C'est un DataFrame 395 x 31
        student_data.shape
```

```
Out[2]: (395, 31)
```

```
In [3]: # Le type de données est un pandas DataFrame
        # Hence I can use pandas DataFrame methods
        type(student_data)
```

```
Out[3]: pandas.core.frame.DataFrame
```

2.1 Mise en uvre : Exploration des données

Commençons par examiner l'ensemble de données pour déterminer le nombre d'étudiants sur lesquels nous avons de l'information et pour connaître le taux d'obtention de diplôme chez ces étudiants. Dans la cellule de code ci-dessous, nous calculerons ce qui suit :

- Le nombre total d'étudiants, n_étudiants.
- Le nombre total de fonctionnalités pour chaque étudiant, n_fonctionnalités.
- Le nombre d'étudiants qui ont réussi, n_passed.
- Le nombre de ces étudiants qui ont échoué, n_failed.
- Le taux de diplomation de la classe, grad_rate, en pourcentage (%)

```
In [24]: ## nombre d'étudiants
         nb_student=student_data.shape[0]
         ## nombre de fonctionnalités
         nb_feat=student_data.shape[1]-1
         ## nombre d'étudiants qui ont réussi
         student_passed=student_data.loc[student_data.passed=="yes","passed"]

         nb_passed=student_passed.shape[0]
         ## nombre d'étudiants qui ont échoué
         student_failed=student_data.loc[student_data.passed=="no","passed"]
         nb_failed=student_failed.shape[0]
         ## taux d'obtention de diplôme
         grad_rate=nb_passed*100/(nb_passed+nb_failed)

         print ("nombre d'étudiants : {}".format(nb_student))
         print("nombre de caractéristiques : {}".format(nb_feat))
         print("nombre d'étudiants réussis  : {}".format(nb_passed))
         print("nombre d'étudiants ayant échoué : {}".format(nb_failed))
         print("Le taux d'obtention du diplôme est de : {:.2f}%".format(grad_rate))
```

```
nombre d'étudiants : 395
nombre de caractéristiques : 30
nombre d'étudiants réussis  : 265
nombre d'étudiants ayant échoué : 130
Le taux d'obtention du diplôme est de : 67.09%
```

2.2 Préparation des données

Dans cette section, nous préparerons les données pour la modélisation, la formation et les essais.

Nous allons identifier les colonnes de caractéristiques et de cibles Il arrive souvent que les données que nous obtenons contiennent des caractéristiques non numériques. Cela peut être un problème, car la plupart des algorithmes d'apprentissage machine s'attendent à ce que les données numériques effectuent des calculs avec.

La cellule de code ci-dessous sépare les données élèves en colonnes de caractéristiques et de cibles afin de voir si certaines caractéristiques sont non numériques.

```
In [16]: # Colonnes
         student_data.columns

Out[16]: Index(['school', 'sex', 'age', 'address', 'famsize', 'Pstatus', 'Medu', 'Fedu',
                'Mjob', 'Fjob', 'reason', 'guardian', 'traveltime', 'studytime',
                'failures', 'schoolsup', 'famsup', 'paid', 'activities', 'nursery',
                'higher', 'internet', 'romantic', 'famrel', 'freetime', 'goout', 'Dalc',
                'Walc', 'health', 'absences', 'passed'],
               dtype='object')

In [6]: #  Nous voulons obtenir le nom de la colonne "passed" qui est le dernier nom de la col
        student_data.columns[-1]

Out[6]: 'passed'

In [7]: #Ceci obtiendrait tout sauf le dernier élément qui est "passed".
        student_data.columns[:-1]

Out[7]: Index(['school', 'sex', 'age', 'address', 'famsize', 'Pstatus', 'Medu', 'Fedu',
               'Mjob', 'Fjob', 'reason', 'guardian', 'traveltime', 'studytime',
               'failures', 'schoolsup', 'famsup', 'paid', 'activities', 'nursery',
               'higher', 'internet', 'romantic', 'famrel', 'freetime', 'goout', 'Dalc',
               'Walc', 'health', 'absences'],
              dtype='object')

In [25]: # Extraire des colonnes de caractéristiques
         # Comme nous l'avons vu plus haut, nous obtenons toutes les colonnes à l'exception de
         featured_col=list(student_data.columns[:-1])
         target_col=student_data.columns[-1]
         # Afficher la liste des colonnes
         print("Colonnes en vedette : \n{}".format(featured_col))
         print("\n Colonne cible : {}".format(target_col))
```

```
        # Séparer les données en données de caractéristiques et données cibles (X_all et y_al
        X_all=student_data[featured_col]
        y_all=student_data[target_col]
        # Afficher les informations sur les caractéristiques en print les cinq premières lign
        print("Informations en vedette : \n{}".format(X_all.head()))
```

```
Colonnes en vedette :
['school', 'sex', 'age', 'address', 'famsize', 'Pstatus', 'Medu', 'Fedu', 'Mjob', 'Fjob', 'reas

 Colonne cible : passed
Informations en vedette :
  school sex  age address famsize Pstatus  Medu  Fedu     Mjob      Fjob  \
0     GP   F   18       U     GT3       A     4     4  at_home   teacher
1     GP   F   17       U     GT3       T     1     1  at_home     other
2     GP   F   15       U     LE3       T     1     1  at_home     other
3     GP   F   15       U     GT3       T     4     2   health  services
4     GP   F   16       U     GT3       T     3     3    other     other

    ...    higher internet  romantic  famrel  freetime goout Dalc Walc health  \
0   ...       yes       no        no       4         3     4    1    1      3
1   ...       yes      yes        no       5         3     3    1    1      3
2   ...       yes      yes        no       4         3     2    2    3      3
3   ...       yes      yes       yes       3         2     2    1    1      5
4   ...       yes       no        no       4         3     2    1    2      5

  absences
0        6
1        4
2       10
3        2
4        4

[5 rows x 30 columns]
```

2.3 Colonnes des caractéristiques de prétraitement

Il y a plusieurs colonnes non numériques qui doivent être converties ! Beaucoup d'entre eux sont simplement oui/non, par exemple Internet. Ceux-ci peuvent être raisonnablement convertis en valeurs de 1/0 (binaires).

D'autres colonnes, comme Mjob et Fjob, ont plus de deux valeurs et sont appelées variables catégorielles. La façon recommandée pour gérer une telle colonne est de créer autant de colonnes que possible (par exemple Fjob_teacher, Fjob_other, Fjob_services, etc.), et d'attribuer un 1 à l'une d'elles et 0 à toutes les autres.

Ces colonnes générées sont parfois appelées variables factices, et nous utiliserons la fonction pandas.get_dummies() pour effectuer cette transformation. La cellule de code ci-dessous exécute la routine de prétraitement décrite dans cette section.

```
In [26]: def preprocess_features(X):
             ''' Prétraite les données élèves et convertit les variables binaires non numériqu
                 variables binaires (0/1). Convertit les variables catégorielles en variables

             # Initialiser le nouveau DataFrame de sortie
             output = pd.DataFrame(index = X.index)

             # Étudier chaque colonne de caractéristiques pour les données
             for col, col_data in X.iteritems():

                 # Si le type de données n'est pas numérique, remplacer toutes les valeurs oui,
                 if col_data.dtype == object:
                     col_data = col_data.replace(['yes', 'no'], [1, 0])

                 # Si le type de données est catégorique, convertir en variables fictives
                 if col_data.dtype == object:
                      col_data = pd.get_dummies(col_data, prefix = col)

                 # Collecter les colonnes révisées
                 output = output.join(col_data)

             return output

         X_all = preprocess_features(X_all)
         print ("Colonnes de caractéristiques traitées ({} caractéristiques totales):\n{}".form
```

```
Colonnes de caractéristiques traitées (48 caractéristiques totales):
['school_GP', 'school_MS', 'sex_F', 'sex_M', 'age', 'address_R', 'address_U', 'famsize_GT3', '
```

2.4 Mise en uvre : Répartition des données de formation et de test

Jusqu'à présent, nous avons converti toutes les caractéristiques catégorielles en valeurs numériques. Pour l'étape suivante, nous divisons les données (à la fois les caractéristiques et les étiquettes correspondantes) en ensembles de formation et de test. Dans la cellule de code suivante ci-dessous, il est implémenté ce qui suit :

- Mélanger aléatoirement et diviser les données (X_all, y_all) en sous-ensembles de formation et de test.
- Utiliser 300 points de formation (environ 75 %) et 95 points de test (environ 25 %).
- Définir un random_state pour la ou les fonctions utilisées, le cas échéant.
- Stocker les résultats dans X_train, X_test, y_train et y_test.

```
In [18]: # TODO : Importez toutes les fonctionnalités supplémentaires dont vous pourriez avoir
         from sklearn.model_selection import train_test_split

In [27]: # Pour le fractionnement initial train/test, nous pouvons obtenir une stratification
         X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, stratify = y_all, te
```

```
        # Afficher les résultats du fractionnement
        print ("L'ensemble d'entraînement a {} échantillons.".format(X_train.shape[0]))
        print ("L'ensemble de test a  {} échantillons.".format(X_test.shape[0]))
```

```
L'ensemble d'entraînement a 300 échantillons.
L'ensemble de test a  95 échantillons.
```

2.5 Modèles de formation et d'évaluation

Dans cette section, on choisira 3 modèles d'apprentissage supervisé qui sont appropriés pour ce problème et disponibles en scikit-learn. On adaptera ensuite le modèle à différentes tailles de données d'entraînement (100 points de données, 200 points de données et 300 points de données) et mesurera le score F1.

2.6 Setup

La cellule de code ci-dessous permet d'initialiser trois fonctions d'aide utilisé pour former et tester les trois modèles d'apprentissage supervisé que vous avez choisis ci-dessus. Les fonctions sont les suivantes :

train_classifier - prend comme entrée un classificateur et des données d'entraînement et adapte le classificateur aux données. predict_labels - prend en entrée un classificateur d'ajustement, des caractéristiques et un étiquetage de cible et fait des prédictions en utilisant le score F1. train_predict - prend en entrée un classificateur, et les données de formation et de test, et exécute train_clasifier et predict_labels. Cette fonction rapportera le score F1 séparément pour les données d'entraînement et d'examen.

```
In [28]: def train_classifier(clf, X_train, y_train):
             ''' Adapte un classificateur aux données d'entraînement. '''

             start = time()
             clf.fit(X_train, y_train)
             end = time()

             print ("Modèle formé en {:.4f} seconds".format(end - start))


         def predict_labels(clf, features, target):
             ''' Fait des prédictions à l'aide d'un classificateur d'ajustement basé sur le sc

             start = time()
             y_pred = clf.predict(features)
             end = time()

             print ("A fait des prédictions dans {:.4f} seconds.".format(end - start))
             return f1_score(target.values, y_pred, pos_label='yes')
```

```
def train_predict(clf, X_train, y_train, X_test, y_test):
    ''' S'entraîner et prédire à l'aide d'un classificateur basé sur le score F1. '''

    print ("")
    print ("L'Entrainement a {} utilisant une taille de l'ensemble de formation de {}

    train_classifier(clf, X_train, y_train)

    print ("Score F1 pour le set d'entraînement: {:.4f}.".format(predict_labels(clf,
    print("Score F1 pour le set de test: {:.4f}.".format(predict_labels(clf, X_test,
```

2.7 Implémentation : Paramètres de performance du modèle

Avec les fonctions prédéfinies ci-dessus, on va maintenant importer les trois modèles d'apprentissage supervisé et exécuter la fonction train_predict pour chacun. Dans la cellule de code suivante, on va implémenter ce qui suit :

- Importer les trois modèles d'apprentissage supervisé.
- Initialiser les trois modèles et les stocker dans clf_A, clf_B et clf_C.
- Utiliser un random_state pour chaque modèle utilisé; le cas échéant. *Remarque* : On utilisera les paramètres par défaut pour chaque modèle -
- Créer les différentes tailles d'ensembles d'entraînement à utiliser pour entraîner chaque modèle.
- Adapter chaque modèle à la taille de chaque ensemble d'entraînement et faire des prédictions sur l'ensemble d'essai (9 au total).

In [29]:
```
# TODO : Importer les trois modèles d'apprentissage supervisé de sklearn
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

# TODO : Initialiser les trois modèles
clf_A = GaussianNB()
clf_B = LogisticRegression(random_state=42)
clf_C = SVC(random_state=42)

# TODO : Configurez les tailles des sets d'entraînement
X_train_100 = X_train.iloc[:100, :]
y_train_100 = y_train.iloc[:100]

X_train_200 = X_train.iloc[:200, :]
y_train_200 = y_train.iloc[:200]

X_train_300 = X_train.iloc[:300, :]
y_train_300 = y_train.iloc[:300]

# TODO : Exécuter la fonction'train_predict' pour chaque classificateur et chaque tai
# train_predict(clf, X_train, y_train, y_train, X_test, y_test)
```

```
for clf in [clf_A, clf_B, clf_C]:
    print("\n{}: \n".format(clf.__class__.__name__))
    for n in [100, 200, 300]:
        train_predict(clf, X_train[:n], y_train[:n], X_test, y_test)
```

```
GaussianNB:


L'Entrainement a GaussianNB utilisant une taille de l'ensemble de formation de 100. . .
Modèle formé en 0.0040 seconds
A fait des prédictions dans 0.0030 seconds.
Score F1 pour le set d'entraînement: 0.7752.
A fait des prédictions dans 0.0020 seconds.
Score F1 pour le set de test: 0.6457.

L'Entrainement a GaussianNB utilisant une taille de l'ensemble de formation de 200. . .
Modèle formé en 0.0030 seconds
A fait des prédictions dans 0.0020 seconds.
Score F1 pour le set d'entraînement: 0.8060.
A fait des prédictions dans 0.0050 seconds.
Score F1 pour le set de test: 0.7218.

L'Entrainement a GaussianNB utilisant une taille de l'ensemble de formation de 300. . .
Modèle formé en 0.0040 seconds
A fait des prédictions dans 0.0020 seconds.
Score F1 pour le set d'entraînement: 0.8134.
A fait des prédictions dans 0.0020 seconds.
Score F1 pour le set de test: 0.7761.

LogisticRegression:


L'Entrainement a LogisticRegression utilisant une taille de l'ensemble de formation de 100. .
Modèle formé en 0.0020 seconds
A fait des prédictions dans 0.0010 seconds.
Score F1 pour le set d'entraînement: 0.8671.
A fait des prédictions dans 0.0030 seconds.
Score F1 pour le set de test: 0.7068.

L'Entrainement a LogisticRegression utilisant une taille de l'ensemble de formation de 200. .
Modèle formé en 0.0060 seconds
A fait des prédictions dans 0.0020 seconds.
Score F1 pour le set d'entraînement: 0.8211.
A fait des prédictions dans 0.0010 seconds.
Score F1 pour le set de test: 0.7391.
```

```
L'Entrainement a LogisticRegression utilisant une taille de l'ensemble de formation de 300. .
Modèle formé en 0.0040 seconds
A fait des prédictions dans 0.0020 seconds.
Score F1 pour le set d'entraînement: 0.8512.
A fait des prédictions dans 0.0010 seconds.
Score F1 pour le set de test: 0.7500.

SVC:


L'Entrainement a SVC utilisant une taille de l'ensemble de formation de 100. . .
Modèle formé en 0.0030 seconds
A fait des prédictions dans 0.0010 seconds.
Score F1 pour le set d'entraînement: 0.8354.
A fait des prédictions dans 0.0020 seconds.
Score F1 pour le set de test: 0.8025.

L'Entrainement a SVC utilisant une taille de l'ensemble de formation de 200. . .
Modèle formé en 0.0060 seconds
A fait des prédictions dans 0.0040 seconds.
Score F1 pour le set d'entraînement: 0.8431.
A fait des prédictions dans 0.0020 seconds.
Score F1 pour le set de test: 0.8105.

L'Entrainement a SVC utilisant une taille de l'ensemble de formation de 300. . .
Modèle formé en 0.0100 seconds
A fait des prédictions dans 0.0060 seconds.
Score F1 pour le set d'entraînement: 0.8664.
A fait des prédictions dans 0.0030 seconds.
Score F1 pour le set de test: 0.8052.


C:\Users\acer\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:433: FutureWarning:
  FutureWarning)
C:\Users\acer\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:433: FutureWarning:
  FutureWarning)
C:\Users\acer\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:433: FutureWarning:
  FutureWarning)
C:\Users\acer\Anaconda3\lib\site-packages\sklearn\svm\base.py:196: FutureWarning: The default v
  "avoid this warning.", FutureWarning)
C:\Users\acer\Anaconda3\lib\site-packages\sklearn\svm\base.py:196: FutureWarning: The default v
  "avoid this warning.", FutureWarning)
C:\Users\acer\Anaconda3\lib\site-packages\sklearn\svm\base.py:196: FutureWarning: The default v
  "avoid this warning.", FutureWarning)
```

3 Conclusion

La performance prédictive des MVC est légèrement supérieure à celle de Naive Bayes. Cependant, il est important de noter que le temps de calcul des MVC augmenterait beaucoup plus rapidement que celui de Naive Bayes avec plus de données, et que nos coûts augmenteraient de façon exponentielle lorsque nous aurions plus d'étudiants. D'un autre côté, le temps de calcul de Naive Bayes augmenterait linéairement avec plus de données, et notre coût n'augmenterait pas aussi rapidement. Par conséquent, Naive Bayes offre une bonne alternative aux MVC en tenant compte de sa performance sur un petit ensemble de données et sur un ensemble de données potentiellement important et croissant.

Par conséquent, nous comparons Naive Bayes et Logistic Regression. Bien que les résultats montrent que la régression logistique est légèrement inférieure à celle de Naive Bayes en termes de performance prédictive, un léger ajustement du modèle de régression logistique permettrait facilement d'obtenir une performance prédictive bien meilleure que celle de Naive Bayes. Cela contraste avec Naive Bayes où nous n'avons pas l'occasion d'accorder le modèle. Par conséquent, nous devrions opter pour la régression logistique.

Notation Big O pour les 3 algorithmes:

- Naive Bayes : O(n)
- Régression logistique : O(Cˆn)
- Support Vector Machines : O(nˆ3) avec noyau sigmoïde ou O(nˆ2) avec des complexités d'espace