Projet sciences des données

Bahrman Louis, Yuxuan Wang

I. Importation des données

On utilise la fonction `read_csv` du module pandas qui renvoie un objet du type `Dataframe` :

```
import pandas as pd
weatherFrame=pd.read_csv('weather.csv', true_values=["Yes"],
false_values=["No"])
```

Ici les options par défaut sont suffisantes. En effet, les séparateurs sont des virgules, et la première ligne contient l'en-tête. Les comme ceux par défaut. Néanmoins, on souhaite transformer les `'Yes'` en `True` et les `'No'` en `False`. Cela est possible grâce aux paramètres `true_values` et `false_values`.

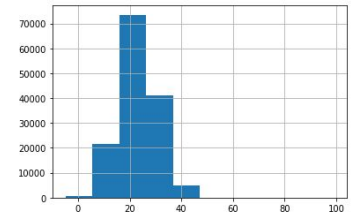
II. Examen des données

On exploite ensuite les attributs de l'objet `weatherFrame` ainsi créé.

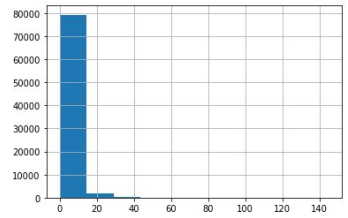
```
print("Nb de lignes et colonnes :
",weatherFrame.shape,"\n","nb d'elements ",weatherFrame.size)
print("type de donnees : ",weatherFrame.dtypes)
weatherFrame['MaxTemp'].hist()
```

On utilise les méthodes `shape, size, dtypes`
On remarque que `weatherFrame.dtypes['RainToday']` montre ces données ne sont pas du type `bool` contrairement à `'RainTomorrow'`, mais du type `Object`. Cela est dû aux valeurs manquantes `nan`.
La méthode `hist` permet d'afficher la répartition des différentes variables. On remarque que la plupart suit une répartition suivant une loi normale. L'échelle horizontale nous permet déjà de suspecter des valeurs aberrantes pour `MaxTemp`, `Evaporation` et `Risk_MM`.

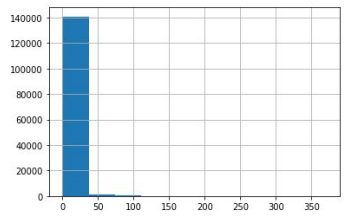
<u>Répartition des données suspectes :</u>



‘MaxTemp’



‘Evaporation’



‘RISK_MM’

III. Préparation des données

On commence par supprimer les valeurs aberrantes pour les remplacer ensuite. On les repère avec les méthodes `min` et `max` de la classe DataFrame. La température maximale de 99.0°C semble être aberrante. Les valeurs hautes de l’évaporations sont quant à elles justifiées par le climat aride de l’Australie (*Luke, G J, Burke, K L, and O'Brien, T M. (1987), Evaporation data for Western Australia. Department of Agriculture and Food, Western Australia, Perth. Report 65.*).

```
weatherFrame['MaxTemp'].max()
```

On doit ensuite supprimer ces données aberrantes. On crée pour cela la fonction `supprimeValeursAberrantes`.

```
def supprimeValeursAberrantes(frame, colonne, min, max):
     for i in range(frame.shape[0]):
     if frame[colonne][i]<min or frame[colonne][i]>max :
          frame[colonne][i]=np.nan #Pour supprimer des
valeurs, on les remplace par np.nan
```

```
supprimeValeursAberrantes(weatherFrame,'MaxTemp',-10,50)
```

Cette fonction supprime en place les valeurs aberrantes d’une `DataFrame` en prenant en paramètres la dataFrame, la colonne à étudier ainsi que le min et le max des valeurs considérées commes normales.

On supprime maintenant les données non pertinentes. La variable ‘`RISK_MM`’ servant à construire la variable cible ‘`RainTomorrow`’, on la supprime en utilisant la méthode `drop`.
Toutes les modifications (y compris celles dans les autres parties) seront faites en place, donc on ajoute le paramètre `inplace=True`.

```
weatherFrame.drop('RISK_MM',axis='columns',inplace=True)
```

On doit maintenant compléter les valeurs manquantes.
Comme conseillé dans l'énoncé, on supprime les colonnes pour lesquelles plus d'⅓ des données sont indisponibles. Pour cela, on utilise la méthode `isna` qui renvoie une `DataFrame` contenant des `1` à la place des valeurs nulles et des `0` à la place des valeurs non nulles, de telle sorte que la somme (méthode `sum`) sur la colonne renvoie le nombre de valeurs nulles:

```
def supprimeColonnesManquantes(frame):
     nbLignes=frame.shape[0]
     for colonne in frame:
     if frame.isna().sum()[colonne]>=nbLignes/3:
          frame.drop(colonne,axis='columns',inplace=True)
```

```
supprimeColonnesManquantes(weatherFrame)
```

Parmi les colonnes restantes, on va compléter les données selon différentes méthodes, selon qu'elles soient qualitatives ou quantitatives.
Pour les données numériques, on calcule leur moyenne, puis on remplace les valeurs manquantes par celle-ci :

```
valeursMedianes=weatherFrame.median(axis=0,skipna=True,numeri
c_only=True)
weatherFrame.fillna(valeursMedianes)
```

Pour les données non numériques, on utilise la méthode de complétion `'pad'`, qui remplace les valeurs nulles par la dernière valeur non nulle lue :

```
def completevaleursQualitativesManquantes(frame):
     for colonne in frame:
     if frame.dtypes[colonne]=='O':

weatherFrame[colonne].fillna(method='pad',inplace=True)
```

```
completevaleursQualitativesManquantes(weatherFrame)
```

On doit ensuite transformer les valeurs quantitatives en valeurs numériques. On utilise la classe `LabelEncoder` fournie par `skelearn.model_selection`. Cette classe manipulant des tableaux numpy, on importe aussi ce module et on convertit chaque colonne de la dataFrame en tableau numpy à l'aide de la méthode

`to_numpy()`. On travaille colonne par colonne afin de ne pas renvoyer une nouvelle dataFrame en injectant ensuite les données dans la frame donnée.

```
import numpy as np
from sklearn.preprocessing import LabelEncoder,
StandardScaler
```

On travaille colonne par colonne afin de ne faire que les modifications sur les colonnes de type `Objet`, c'est à dire les chaînes de caractère.

```
def quantitatif_to_numerique(frame):
     for colonne in frame:
     if frame.dtypes[colonne]=='O':
          lc=LabelEncoder()
          colonne_np=frame[colonne].to_numpy()
          frame[colonne]=lc.fit_transform(colonne_np)
```

On doit ensuite normaliser les variables en utilisant la méthode `fit_transform` de la classe `StandardScaler`. On doit ici encore travailler sur des objets numpy en définissant une variable temporaire `frameNp` qui contient l'ensemble de la frame formatée comme tableau `numpy`. On doit ensuite renvoyer une `DataFrame` puisqu'on devra utiliser des méthodes de cette classe pour la recherche de corrélations. On ne fait donc plus de modifications en place pour la fonction `recalibrage`.

```
def recalibrage(frame):
     sc=StandardScaler()
     frameNp=frame.to_numpy()
     sc.fit(frameNp)

frame=pd.DataFrame(sc.transform(frameNp,copy=False),columns=f
rame.columns)
     return frame
```

```
weatherFrame=recalibrage(weatherFrame)
```

IV. Recherche de corrélations

Afin de déterminer les corrélations, on utilise la méthode `corr` de la classe `DataFrame`. Cette fonction renvoie une dataframe contenant les coefficients de corrélation entre les différentes variables.
Le coefficient de corrélation de deux variables X et Y mesure le degré de dépendance linéaire de X et Y. Il est défini par $r(X,Y)=\frac{Cov(X,Y)}{\sigma(X)\cdot\sigma(Y)}$, où

$Cov(X,Y) = \sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$. C'est ce coefficient qui est utilisé par la méthode `'pearson'`, paramètre par défaut de `corr`.

On remarque que les variables les plus fortement corrélées sont `'MinTemp'` et `'MaxTemp'` (coefficient de corrélation de 0.73) ainsi que toutes les données numériques prises à 9am et 3pm qui sont deux à deux corrélées à plus de 65%.

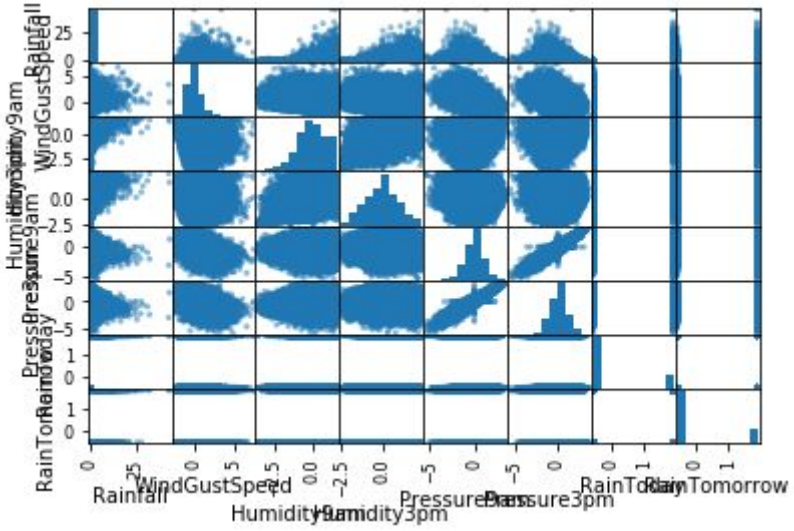
La variable pertinente étant ici `'RainTomorrow'`, on s'intéresse plus particulièrement à la corrélation entre chacune des variables et celle-ci afin de déduire quelles variables sont utiles. On peut automatiser la sélection des variables pertinentes en définissant la fonction `supprimeDonneesNonCorrelees` qui admet en paramètre la colonne cible et le seuil à partir duquel on considère les données comme corrélées. Si les colonnes ne sont pas corrélées, on les supprime en place.

```
def supprimeDonneesNonCorrelees(frame,colonneAComparer,
seuil):

MatriceCorrelations=frame.corr(method='pearson')[colonneAComp
arer]
     for colonne,valeur in MatriceCorrelations.iteritems():
         if abs(valeur)<seuil:
              frame.drop(colonne,axis='columns',inplace=True)
```

```
supprimeDonneesNonCorrelees(weatherFrame,'RainTomorrow',0.2)
```

On peut aussi utiliser la fonction `scatter_matrix` du module `pandas`, même si elle ne permet pas d'effacer les colonnes inutiles.



V. Extraction des jeux d'apprentissage et de test

On doit créer deux jeux, un d'apprentissage et un de test. On utilise la fonction `train_test_split` du module `sklearn.model_selection`. On répartit les

ligne à 75% dans les données d'apprentissage et à 25% dans les données de test, de façon à avoir les meilleurs résultats, grâce à cette fonction `train_test_split`. On sépare les données cibles des autres en utilisant la fonction `extraction` définie ci-dessous :

```
def extraction(frame):
     X_data=frame.iloc[:,:-1].values
     y_data=frame.iloc[:,-1].values
     y_data=np.where(y_data==True, 1,0)
     return train_test_split(X_data,y_data,train_size=0.75
)
```

```
X_train,X_test,Y_train,Y_test=extraction(weatherFrame)
```

VI. Entraînement du modèle

Nous voulons entraîner le modèle à l'aide de la classe `LogisticRegression`. Il s'agit ici d'une régression linéaire binaire, dont la sortie y peut prendre deux valeurs 0 ou 1, mais l'entrée x est continue.

- Notations: si on note p(y=$\frac{1}{x}$)=μ(x), on peut en déduire de la loi des probabilités totales: p(y=$\frac{0}{x}$)=1-μ(x). L'expression de logarithme du rapport des vraisemblances est donc log($\frac{p(y=\frac{1}{x})}{p(y=\frac{0}{x})}$)=log($\frac{\mu(x)}{(1-\mu(x)}$).
- Hypothèses: On fait l'hypothèse que la fonction logit est une fonction linéaire de x:
  logit=$w_0 + w^T x$
- Algorithme: Pour maximiser la vraisemblance,donc minimiser la fonction J(w, w_0) = - log-vraisemblance, nous pouvons utiliser une descente de gradient.
- Paramètres : Ce sont les paramètres w , w_0 dans la fonction logit qui doivent être mis à jour lors de la phase d'apprentissage

Le code qu'on utilise pour réaliser cet algorithme est le suivant :

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=1.0, penalty='l1', tol=0.01)
lr.fit(X_train,Y_train)
lr_y_predict = lr.predict(X_test)
```

Où on utilise la fonction `LogisticRegression()` pour notre jeu d'apprentissage: `lr.fit(X_train,Y_train)`, puis on fait une prédiction: `lr_y_predict = lr.predict(X_test)`.

VII. Évaluation du modèle

Après l'entraînement du modèle, on peut maintenant l'utiliser pour prédire les valeurs de la variable de sortie. Pour cela, on veut évaluer sa performance en comparer les valeurs prédites aux valeurs réelles, à l'aide des différentes métriques de Scikit-Learn:

-`accuracy_score` : le pourcentage de valeurs de sortie prédites qui coïncide avec les vraies valeurs. Pour notre modèle on a accuracy= 0.837.

-`confusion_matrix` : une matrice[2,2] dont les cases ont pour signification:
`C[0,0]` : True Negatives (response 0, predicted 0)
`C[0,1]` : False Positives (response 0, predicted 1)
`C[1,0]` : False Negatives (response 1, predicted 0)
`C[1,1]` : True Positives (response 1, predicted 1)
Pour notre modèle, cela renvoie [[3514, 4450], [1333, 26252]]

-`precision_score`: $\frac{TP}{TP=FP}$ (TP=le nombre de True Positive, FP=le nombre de False Positive) Pour notre modèle on a precision=0.8550583023907238

-recall_score: $\frac{TP}{TP=FN}$ (FN=le nombre de False negative). Pour notre modèle on a recall=0.9516766358528186

-f1: $\frac{2 * precision * recall}{precision + recall}$. Pour notre modèle f1=0.901.

En conclusion, notre modèle a relativement une bonne performance, comme le taux des prédictions réussies est d'environ 83.7% (accuracy score), les valeurs de precision_score, recall_score et f1 sont aussi assez élevées.

VIII. Amélioration de l'évaluation

On cherche ici des possibilité d'amélioration de notre modèle. Pour cela, on peut utiliser la validation croisée, réalisé avec la classe KFold, qui consiste à diviser le jeu d'apprentissage en plusieurs sous-ensemble, puis évaluer à nouveau la performance de la prédiction dans ces sous-ensemble.

```
from sklearn.model_selection import KFold,cross_val_score
kf = KFold(n_splits=5,shuffle=True)
print(cross_val_score(lr,X_data,y_data,
cv=5,scoring='accuracy'))
```

Cela nous donne, pour les 5 sous-ensemble divisées, leur :[0.83182138, 0.8216182 , 0.83870174, 0.83708418, 0.84017863]. Cela montre que notre modèle est assez performant dans les différentes sous-ensembles, puisque la moyenne est de 0.838, ce qui est proche des valeurs obtenues sans validation croisée.