Documentation technique – Dataset Flowers

Louis ADAM / Yusuf FIDAN

Table des matières

1) Généralités ***3***

2) Nettoyage des données ***4***

3) Algorithme de prédiction ***6***

4) Neo4j ***7***

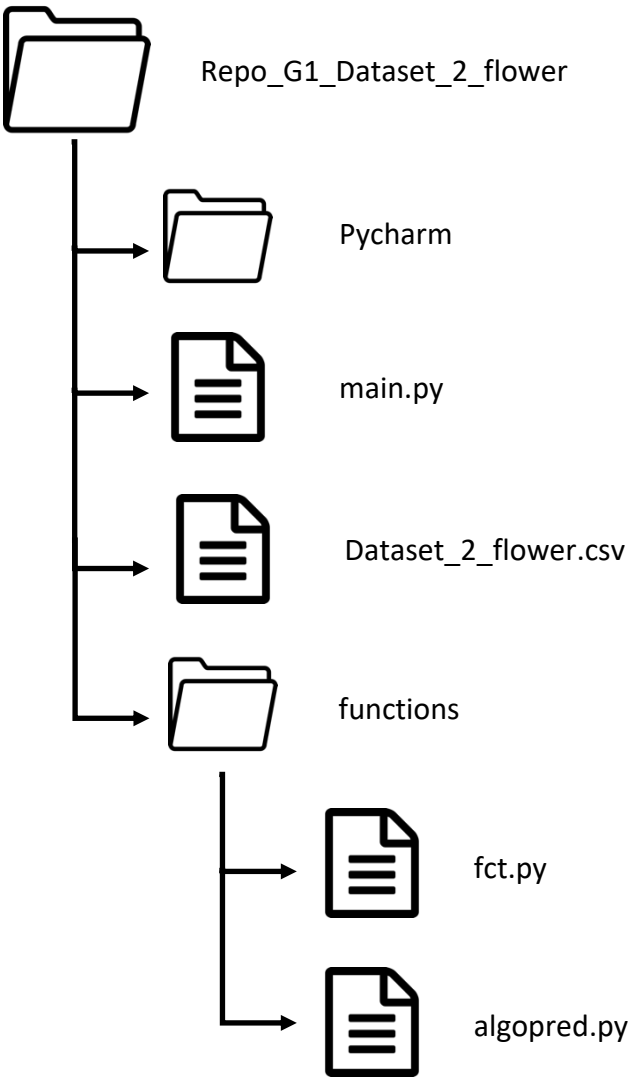
5) Docker ***8***

Aperçu du dataset :

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	66	6.7	3.1	4.4	1.4	Iris-versicolor
1	57	6.3	3.3	4.7	1.6	Iris-versicolor
2	24	5.1	3.3	1.7	0.5	Iris-setosa
3	27	5.0	3.4	1.6	0.4	Iris-setosa
4	143	5.8	2.7	5.1	1.9	Iris-virginica

1) Généralités

Structure du code :



2) Nettoyage des données

Avant de nettoyer les données, il faut exporter le csv en dataframe, à l'aide de la fonction dfcsv :

```
def dfcsv(chemin,separ): #Fonction qui permet de lire un csv et le convertir en dataframe
    """

    :param chemin: string(url)
    :param separ: pandas.dataframe
    :return: dataframe
    """
    dataframe = pd.read_csv(chemin, sep=separ)

    return dataframe
```

Dans ce dataset, les colonnes à exploiter sont **Id**, **SepalLengthCm**, **SepalWidthCm**, **PetalLengthCm**, **PetalWidthCm** et **Species**.

Il faut donc retirer les colonnes non exploitables : **index**, **Unnamed: 0**, **Unnamed: 0.1**, **Unnamed: 0.1.1**, **level_0**, **Unnamed: 0.1.1.1**

Ces colonnes sont retirées à l'aide de la fonction dropColumns :

```
def dropColumns(dataframe, liste): #Permet de supprimer les colonnes inutiles
    """

    :param dataframe: pandas.dataframe
    :param liste: string
    :return: dataframe
    """
    logging.info("Fct dropColumns")
    col=[]
    for i in liste:
        col.append(i)
    logging.info("Contenu de la variable col")
    logging.info(col)
    dataframe=dataframe.drop(col, axis=1) # Supprime les colonnes présentes dans liste
    return dataframe
```

Une fois ces colonnes retirées, il faut analyser les colonnes non propres. On peut les analyser avec la fonction counts. Elle permet de voir les valeurs de chaque colonne afin de repérer les données non exploitables :

```
def counts(dataframe, liste): #Fonction qui permet de verifier le contenu des champs du dataframe
    """

    :param dataframe: pandas.dataframe
    :param liste: string
    :return:
    """
    logging.info("Fct counts")
    for i in liste:
        countColumn = dataframe[i].value_counts() # Permet d'analyser les données des colonnes de la liste.
        print(countColumn)
    logging.info("Contenu de la liste"+liste)
    return
```

Une fois les colonnes analysées, on peut traites les colonnes non propres. Dans ce dataset, seule la colonne SepalLengthCm est inexploitable. Pour y remédier, on va exécuter la fonction cleanCol :

```
def cleanCol(dataframe, colonne): #Fonction qui permet de nettoyer la ou les colonnes exploitables
    """

    :param dataframe: pandas.Dataframe
    :param colonne: string
    :return: dataframe
    """
    logging.info("Fct cleanCol")
    dataframe.loc[(dataframe[colonne] == 'None'), colonne] = math.nan #remplace les None par NaN dans les colonnes saisies
    dataframe=dataframe.astype({colonne: float}) # Convertit tout en float
    moy = dataframe[colonne].mean() # Avoir la moyenne de la colonne
    dataframe[colonne] = dataframe[colonne].fillna(moy) # Remplace les Nan par la moyenne
    logging.info("Moyenne :")
    logging.info(moy)
    return dataframe
```

Cette fonction permet de remplacer les valeurs erronées par la moyenne des valeurs exploitables de la colonne.

Pour la colonne Id, la fonction cleanId permet de faire en sorte que chaque id de fleur soit unique :

```
def cleanId(dataframe,colonne): #Fonction qui permet de rendre la colonne Id exploitable et pertinente
    """

    :param dataframe: pandas.Dataframe
    :param colonne: liste
    :return: dataframe
    """

    for i in dataframe.axes[0]:
        dataframe[colonne][i] = i + 1
    dataframe=dataframe.astype({colonne: int})
    logging.info("Fct cleanId")
    return dataframe
```

Le dataset est donc bien nettoyé. Pour faire une dernière vérification la fonctions isNan permet de générer un dataset contenant les lignes où un nan est présent :

```
def isNan(dataframe): #Fonction qui permet de voir les lignes contenant Nan
    """

    :param dataframe: pandas.Dataframe
    :return:
    """

    is_NaN = dataframe.isnull()
    row_has_NaN = is_NaN.any(axis=1)
    rows_with_NaN = dataframe[row_has_NaN]
    isempty = rows_with_NaN.empty
    print(rows_with_NaN)
    print("Le dataset ne contient pas de Nan :", isempty)
    return
```

3) Algorithme de prédiction

Les algorithmes de prédiction utilisés dans ce projet permettent d'avoir un indice de fiabilité. 2 algorithmes de prédiction sont utilisés dans le code source : le Random Forest et la Régression Logistique. Ces 2 algorithmes sont définis dans la fonction algodf :

```
def algodf(colx,coly,tsize,choixalgo,nesti):

    # Sépare de dataset en 2 sets training et test
    x_train, x_test, y_train, y_test = train_test_split(colx, coly, test_size=tsize)  #Proportion de test et train
    if choixalgo == "rf":
        clf = RandomForestClassifier()
        param_grid = {'n_estimators': nesti} #Nb d'arbres dans le Random Forest
        search = GridSearchCV(clf, param_grid,verbose=1)
        search.fit(x_train, y_train)
    elif choixalgo == "lg":
        clf = LogisticRegression()

    y_pred = search.predict(x_test)

    print("Accuracy {:.10f}".format(accuracy_score(y_test, y_pred)))
    return
```

Le choix de l'algorithme se fait dans l'appel de la fonction (« rf » pour le Random Forest et « lg » pour la Régression Logistique).

Pour le Random Forest, le nombre d'arbre est défini via un GridSearch, qui permet de détecter le paramètre le plus efficace pour avoir le meilleur résultat possible :

```
listeesti=[10, 100, 1000] #GridSearch sur la variable n_estimators (nb d'armes)
algopred.algodf(colrfx, colrfy, 0.3, "rf",listeesti) #Exécution de la fonction algodf
```

Le résultat affiché correspond à l'indice de fiabilité de l'algorithme : plus il se rapproche de 1, plus l'algorithme est fiable.

4) Neo4j

Neo4j est un système de gestion de base de données NOSQL basé sur les graphes.

Neo4j est utilisé dans ce projet afin d'avoir une représentation relationnelle entre les différents champs du dataset.

Pour ce faire, l'utilisation du module py2neo est requis.

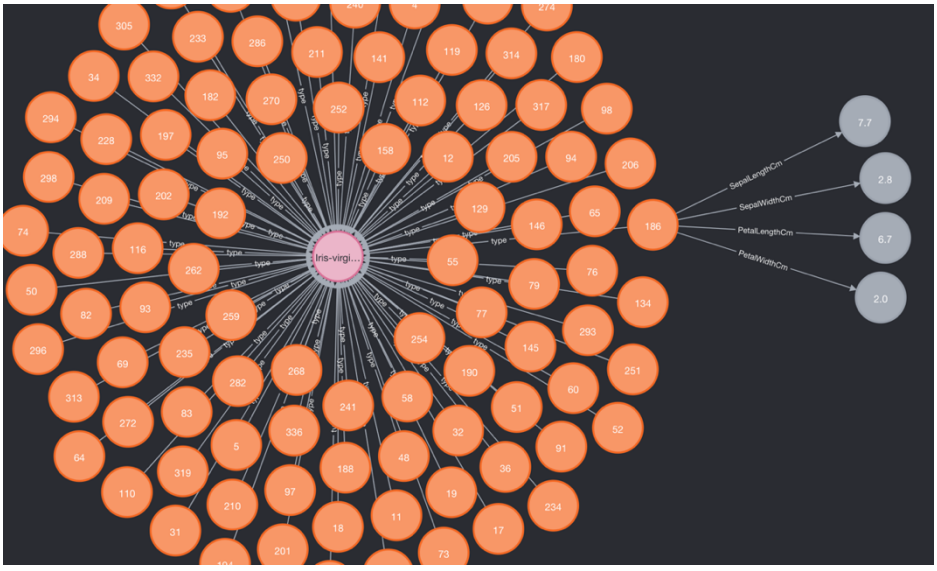
Voici un aperçu du code permettant de faire les relations entre les Id des fleurs, leur type et leur taille :

```
dictCm = { #Dictionnaire contenant les champs Cm ainsi que leur position dans le dataset
    "SepalLengthCm": 1,
    "SepalWidthCm": 2,
    "PetalLengthCm": 3,
    "PetalWidthCm": 4
}

lstspec=[]
for i in range(len(dataframe)): #Parcourt chaque ligne du dataframe
    nodeId=Node("Id", name=i) #Créé un noeud pour chaque ligne
    if (dataframe.iloc[i,[5]]).values == "Iris-versicolor": #Si la fleur est de type versicolor
        if "Iris-versicolor" not in lstspec: #Si iris-versicolor est dans lstspec
            lstspec.append("Iris-versicolor") #On ajoute le Species dans lstspec
            nodeVersi = Node("Species", name="Iris-versicolor") #Créé un noeud Iris-Versicolor
            relIdVersi = Relationship(nodeId, "type", nodeVersi) #Fait la relation entre l'id de la fleur et le type de la fleur
            graphFlower.create(relIdVersi)
        elif "Iris-versicolor" in lstspec: #Si Iris-versicolor n'est pas dans lstspec, pas besoin de recréer le noeud
            relIdVersi = Relationship(nodeId, "type", nodeVersi) #On créé seulement la relation
        graphFlower.create(relIdVersi)
        fct.createGraph(dataframe,nodeId,i,dictCm)

    elif (dataframe.iloc[i,[5]]).values == "Iris-setosa": #Si la fleur est de type setosa
        if "Iris-setosa" not in lstspec: #Si iris-setosa est dans lstspec
            lstspec.append("Iris-setosa") #On ajoute le Species dans lstspec
            nodeSeto = Node("Species", name="Iris-setosa") #Créé un noeud Iris-setosa
            relIdSeto = Relationship(nodeId, "type", nodeSeto) #Fait la relation entre l'id de la fleur et le type de la fleur
```

Voici le résultat :



5) Docker

Une fois le code terminé, il faut mettre tout cela sur un container Docker. Le but est donc de récupérer l'image Neo4j officielle et d'exécuter les commandes nécessaires à l'exécution du script Python via un Dockerfile.
Voici le contenu du Dockerfile :

```
FROM neo4j

RUN bash
RUN apt update -y
RUN apt install python3 -y
RUN apt install python3-pip -y
RUN apt install vim -y
RUN pip3 install pandas
RUN pip3 install sklearn
RUN pip3 install py2neo
RUN pip3 install --upgrade py2neo
RUN mkdir -p /usr/src/app
COPY . /usr/src/app
```

Pour résumer, on utilise l'image officielle neo4j, et on exécute les commandes permettant d'installer python, pip, ainsi que les modules pandas, sklearn, py2neo via pip3.
On crée ensuite un dossier pour y accueillir le code source via la commande COPY.

Une fois le Dockerfile créé, on exécute la commande dans le dossier ou se trouve le Dockerfile :

```
docker build . -t « neo4jmage »
```

L'image se créée, ensuite il faut créer le conteneur depuis l'image précédemment créée :

```
docker run \
   --name containerneo4j \
   -p7474:7474 -p7687:7687 \
   -d \
   -v $HOME/neo4j/data:/data \
   -v $HOME/neo4j/logs:/logs \
   -v $HOME/neo4j/import:/var/lib/neo4j/import \
   -v $HOME/neo4j/plugins:/plugins \
   --env NEO4J_AUTH=none \
   neo4jmage:latest
```

Pour exécuter le code source, on va utiliser la commande docker exec :

```
docker exec -it containerneo4j python3 /usr/src/app/pycharm/main.py
```

Le conteneur est donc prêt et le script est exécuté.