Rapport de Test
Data X - Classification d'items

YANG Fu

A l'attention de monsieur Florian.BRETON

I. Récupération et compréhension de la table “items”

Dans cette partie, nous allons récupérer les informations nécessaires pour comprendre et nettoyer des données.

Code pour extraire les enregistrements dans “df1”

```
#Importation des librairies nécessaires
import psycopg2
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

# Connexion à la base de données
conn = psycopg2.connect(
    host="prod-rds-db.cbijryjiwdgw.eu-west-3.rds.amazonaws.com",
    user="testdata",
    password="testData678341A",
    database="prodkbdb",
    port=5432
)
# Extraction des données
query1 = "SELECT  * FROM items"
df1 = pd.read_sql(query1, conn)
print(df1.head())
```

En utilisant la librairie Pandas, nous avons réussi à récupérer les données ci-dessous.

Tête du contenue de la table “items”

```
                                     id  amount description  date  \
0  95622ac5-f517-4394-9691-830c1d72d4d7    0.00        None  None
1  97418c24-8c15-4055-9259-c8af699c68fe   13.95        None  None
2  401d801d-6aa0-4eda-b387-6c739b3da293    8.60        None  None
3  68a5b978-0547-4af9-9829-5417a8c76c7b    3.02        None  None
4  3633ca34-5beb-42a8-a410-b1dd545f80d8    2.33        None  None

                      itemName parent quantity  taxAmount taxDescription  \
0  goodbye california - medium   None     None       0.00        TVA 0 %
1           formule boeuf 🥑🍟🍷    None     None       1.17        TVA 0 %
2                   pita boeuf   None     None       0.78       TVA 10 %
3                     frites 🍟    None     None       0.27       TVA 10 %
4             coca cherry 33cl   None     None       0.12      TVA 5.5 %

   type                                storeId  \
0  None  f2eb401d-585d-410f-aff9-e35f67ee9e2d
1  menu  897a3d79-b358-4a76-b9bb-f72c857a5565
2  None  897a3d79-b358-4a76-b9bb-f72c857a5565
3  None  897a3d79-b358-4a76-b9bb-f72c857a5565
4  None  897a3d79-b358-4a76-b9bb-f72c857a5565

                        createdAt                        updatedAt  taxRate
0 2023-04-05 17:36:33.471000+00:00 2023-04-05 17:36:33.471000+00:00        0
1 2023-04-05 17:36:34.006000+00:00 2023-04-05 17:36:34.006000+00:00        0
2 2023-04-05 17:36:34.011000+00:00 2023-04-05 17:36:34.011000+00:00     1000
3 2023-04-05 17:36:34.011000+00:00 2023-04-05 17:36:34.011000+00:00     1000
4 2023-04-05 17:36:34.011000+00:00 2023-04-05 17:36:34.011000+00:00      550
```

Dans la table "items", chaque enregistrement correspondant à un "itemName" possède une catégorie associée. Par exemple, pour l'item "**formule boeuf**", sa catégorie est "**menu**".

Nous allons ensuite extraire les données utiles pour la classification : “itemName” et “type” en utilisant le code ci-dessus :

```
# Extraction des données
query = "SELECT  \"itemName\" AS description, type AS category FROM items"
df = pd.read_sql(query, conn)
```

Nous avons remarqué la présence de champs "None" et de dates dans la catégorie de données, ce qui peut potentiellement entraîner des erreurs lors de la fixation du modèle. En supprimant les anomalies, les catégories uniques présentes dans la table sont : "**menu**", "**champ vide**" et "**dish**".

Pour cela, nous allons appliquer trois méthodes de classification (Régression logistique multinomiale, arbres de décision et réseau de neurones récurrents), afin de classifier ces enregistrements (information de type "string") selon leur item.

II. Régression logistique multinomiale

1) Modèle simple

La régression logistique estime la probabilité qu'un événement se produise, tel que voter ou ne pas voter, sur la base d'un ensemble de données donné de variables indépendantes. Cette démarche est également efficace et relativement rapide.

Dans notre cas, les catégories uniques sont "menu", "champ vide" et "dish". Car la variable dépendante (item) a trois résultats possibles, nous allons donc utiliser une régression logistique multinomiale dans un premier temps.

Nous allons ensuite utiliser la classe **TfidfVectorizer** (Term Frequency-Inverse Document Frequency), qui est souvent utilisée afin de convertir les textes en vecteurs numériques (ou matrices).

Cette méthode TFIDF prend en compte l'importance de chaque mot dans un document avec deux facteurs : fréquence du terme (le mot qui apparaît le plus fréquemment) et fréquence inverse du terme (le mot qui apparaît le moins fréquemment). Cela nous permettra d'utiliser l'apprentissage automatique.

Pour assurer une cohérence de modèle, nous avons fixé une proportion de 80% de données d'entraînement et de 20% de données de test.

Code de la régression logistique multinomiale

```
# Séparation des données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(df['description'], df['category'], test_size=0.2, random_state=42)

# Création d'un modèle de classification en utilisant la régression logistique
vectorizer = TfidfVectorizer()
X_train_vectors = vectorizer.fit_transform(X_train)
X_test_vectors = vectorizer.transform(X_test)

# Entraîner le modèle
model = LogisticRegression()
model.fit(X_train_vectors, y_train)

# Évaluation du modèle
y_pred = model.predict(X_test_vectors)
print(classification_report(y_test, y_pred))
```

Résultat du modèle

	precision	recall	f1-score	support
	1.00	0.99	1.00	375588
dish	0.92	1.00	0.96	1608
menu	0.94	0.94	0.94	27868
accuracy			0.99	405064
macro avg	0.95	0.98	0.97	405064
weighted avg	0.99	0.99	0.99	405064

Nous remarquons que le modèle obtenu a une performance élevée. De plus, nous avons noté que le temps d'exécution est de 55 secondes, ce qui est relativement rapide par rapport aux tests suivants. Plus précisément, les valeurs de la précision, recall et f1-score sont élevées.

2) Modèle en appliquant la méthode SMOTE

Cependant, le modèle précédent n'a pas convergé et a atteint la limite du nombre maximum d'itérations. La répartition des trois classes nous a également alerté qu'il y a un déséquilibre entre ces trois classes, avec un grand nombre de données manquantes (champs vides) et un petit nombre de données "dish". Il semble que le modèle est suréchantillonnage.

Nous allons ensuite appliquer la méthode **SMOTE**, qui s'adresse à gérer le problème de la classe minoritaire afin d'éviter le problème de suréchantillonné.

Code avec ajout de la méthode SMOTE :

```
# Appliquer SMOTE pour équilibrer les classes
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train_vectors, y_train)
```

Résultat après l'amélioration

	precision	recall	f1-score	support
	1.00	0.98	0.99	378819
dish	0.91	1.00	0.95	1616
menu	0.82	0.99	0.90	28112
accuracy			0.98	408547
macro avg	0.91	0.99	0.95	408547
weighted avg	0.99	0.98	0.98	408547

Pour cela, nous remarquons que le résultat présent est plus cohérent que le modèle précédent avec une précision de 0.98. Néanmoins, nous avons encore rencontré un problème de convergence (l'itération maximum a été atteinte). Nous allons donc essayer d'augmenter le nombre d'itérations.

3) Modification du nombre d'itérations

Ici, nous avons rajouté un paramètre : **`max_iter=1000`** afin d'éviter le problème de convergence.

	precision	recall	f1-score	support
	1.00	0.98	0.99	379361
dish	0.91	1.00	0.95	1648
menu	0.81	0.99	0.89	28189
accuracy			0.98	409198
macro avg	0.91	0.99	0.95	409198
weighted avg	0.99	0.98	0.98	409198

Cette fois-ci, nous avons éliminé le problème de non-convergence, avec un modèle plus cohérent par rapport aux deux modèles précédents. Cependant, le temps d'exécution devient plus long avec 14 minutes 19 secondes, environ 14 fois plus long que le modèle simple.

Comparaison entre ces trois modèle :

Modèle	Temps d'exécution	Précision
Modèle simple	55 secondes	0,99
Modèle en appliquant la méthode SMOTE	5 minutes 33 secondes	0,98
Modèle avec variation du nombre d'itérations	14 minutes 19 secondes	0,98

Code du meilleur modèle en utilisant la régression logistique multinomiale :

```python
from imblearn.over_sampling import SMOTE
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split

# Séparation des données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(df['description'], df['category'], test_size=0.2, random_state=42)

# Vectorisation des données textuelles
vectorizer = TfidfVectorizer()
X_train_vectors = vectorizer.fit_transform(X_train)
X_test_vectors = vectorizer.transform(X_test)

# Appliquer SMOTE pour équilibrer les classes
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train_vectors, y_train)

# Entraîner le modèle avec les données équilibrées
model = LogisticRegression(max_iter=1000)
model.fit(X_train_res, y_train_res)

# Évaluation du modèle
y_pred = model.predict(X_test_vectors)
print(classification_report(y_test, y_pred))
```

Fonction renvoyant la catégorie associée à cet "item"

```python
# Fonction de classification
def classify_item_logistique(item):
    item_vector = vectorizer.transform([item])
    category = model.predict(item_vector)
    return category[0]
```

Conclusion : Le meilleur modèle que nous avons trouvé est le modèle en modifiant le nombre d'itérations et en équilibrant la classe minoritaire.

III. Arbres de décision

Dans cette partie, nous allons essayer la méthode d'arbres de décision. Les avantages des arbres de décision sont leur capacité à gérer des données non linéaires et à prendre en compte des caractéristiques complexes. Nous allons donc classifier les tickets de caisse avec cette méthode.

De même, nous utilisons la méthode TFIDF (Term Frequency-Inverse Document Frequency) afin de déterminer l'importance des mots.

Code du modèle :

```
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report
from imblearn.over_sampling import SMOTE

# Séparation des données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(df['description'], df['category'], test_size=0.2, random_state=42)

# Création d'un modèle d'arbre de décision
vectorizer = TfidfVectorizer()
X_train_vectors = vectorizer.fit_transform(X_train)
X_test_vectors = vectorizer.transform(X_test)

# Appliquer SMOTE pour équilibrer les classes
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train_vectors, y_train)

model = DecisionTreeClassifier()
model.fit(X_train_res, y_train_res)

# Évaluation du modèle
y_pred = model.predict(X_test_vectors)
print(classification_report(y_test, y_pred))
```

Résultat obtenu avec un temps d'exécution de 12 mins 22s :

	precision	recall	f1-score	support
	1.00	0.99	0.99	379361
dish	0.91	1.00	0.96	1648
menu	0.89	0.99	0.94	28189
accuracy			0.99	409198
macro avg	0.94	0.99	0.96	409198
weighted avg	0.99	0.99	0.99	409198

Nous avons observé que les valeurs de précision, recall et f1-score sont relativement plus grandes que le modèle précédent, ce qui montre que ce modèle a une performance élevée de classifier les textes.

Code de la fonction de classification :

```python
# Fonction de classification
def classify_item_arbres(item):
    item_vector = vectorizer.transform([item])
    category = model.predict(item_vector)
    return category[0]
```

Conclusion : l'arbre de décision est plus performant que la régression linéaire.

IV. Réseau de neurones récurrents

Dans cette partie, nous allons appliquer la méthode de réseau de neurones récurrents pour classifier les catégories de ticket de caisse.

De même, nous cherchons une méthode pour convertir les textes en valeur numérique dans un premier temps. Pour RNN, nous avons la classe Tokenizer, qui construit un index des mots présents dans le texte et attribue un identifiant unique à chaque mot. Dans notre cas, nous avons observé que la longueur des items (données X) est différente, il est donc nécessaire de les ajuster pour les rendre de même longueur. Cela favorise le traitement des lots lors de l'entraînement du modèle.

Une fois que les données entrantes sont converties en séquences d'entiers, nous pouvons commencer à entraîner notre modèle de réseau de neurones. Pour les données Y, nous avons utilisé la méthode LabelEncoder pour convertir en valeurs numériques.

Nous ajoutons une valeur 1 pour avoir le nombre total de mots étant donné que l'indexation commence par 0.

Pour la première couche, nous utilisons une couche d'embedding au modèle, ce qui permet de représenter les mots en tant que vecteurs denses dans un espace continu.

Ensuite, nous rajoutons la deuxième couche LSTM (Long Short-Term Memory) au modèle, qui peut capturer les dépendances à long terme dans les séquences. Généralement, nous choisissons une petite valeur comme 32 ou 64 pour le modèle de RNN. Cependant, nous mettons un paramètre plus grand (128) puisque le problème est beaucoup plus compliqué avec un grand nombre de données.

La fonction d'activation softmax est utilisée dans notre cas pour convertir les sorties du modèle en probabilités pour la classification de 3 dimensions, facilitant ainsi l'interprétation des prédictions et le calcul des métriques de performance.

Code de RNN :

```
# Prétraitement des données
tokenizer = Tokenizer()
tokenizer.fit_on_texts(df['description'])
sequences = tokenizer.texts_to_sequences(df['description'])
X = pad_sequences(sequences)

label_encoder = LabelEncoder()
y = label_encoder.fit_transform(df['category'])

# 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=0.2, random_state=42)

# Création du modèle RNN
vocab_size = len(tokenizer.word_index) + 1
embedding_dim = 100

model = Sequential()
model.add(Embedding(vocab_size, embedding_dim, input_length=X.shape[1]))
model.add(LSTM(128))
model.add(Dense(len(label_encoder.classes_), activation='softmax'))

model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Entraînement du modèle
model.fit(X_train, y_train, epochs=5, batch_size=32, validation_data=(X_test, y_test))

# Évaluation du modèle
_, accuracy = model.evaluate(X_test, y_test)
print('Accuracy:', accuracy)
```

Le résultat obtenu : pour un epochs, nous avons obtenu une valeur de loss faible et une valeur hausse de précision (val_loss = 0.0104, val_accuracy = 0.9967). Cependant , le temps d'exécution est trop long de 40 minutes pour un epochs. Nous considérons donc que ce modèle est plus coûteux que les modèles précédents.

V. Conclusion

La classification des articles trouvés dans les tickets de caisse peut être réalisée en utilisant des méthodes de machine learning, comme la régression linéaire, les arbres de décision ainsi que les réseaux de neurones.
La régression linéaire et les arbres de décision présentent tous deux de bonnes performances avec des temps d'exécution relativement courts. Cependant, les arbres de décision surpassent légèrement la méthode de régression linéaire en termes de performance.
Pour les réseaux de neurones, nous remarquons que les temps d'exécution sont légèrement plus longs que pour les deux premières méthodes. Pour cela, les arbres de décision montrent une meilleure performance en prenant en compte ces deux facteurs : temps d'exécution et précision.

Finalement, nous avons donc choisi le modèle d'arbres de décision pour créer une fonction de classification.