25/11/2021

RAPPORT PROJET AML

AMINATA SEYDI, MAXIME LOUWARD, NADA MATROUF

Table des matières

I. Analyse du dataset

Introduction au problème

Le but du problème est de prédire la valeur de *transactionRevenue* pour un visitorId donné. Pour cela nous disposons de ces données pour un client :

Nom colonnes	Description
fullVisitorId	A unique identifier for each user of the Google Merchandise Store.
channelGrouping	The channel via which the user came to the Store.
device	The specifications for the device used to access the Store.
geoNetwork	This section contains information about the geography of the user.
socialEngagementType	Engagement type, either "Socially Engaged" or "Not Socially Engaged".
totals	This section contains aggregate values across the session.
trafficSource	This section contains information about the Traffic Source from which the session originated.
visitId	An identifier for this session. This is part of the value usually stored as the _utmb cookie. This is only unique to the user. For a unique ID, you should use a combination of fullVisitorId and visitId.
visitNumber	The session number for this user. If this is the first session, then this is set to 1.
visitStartTime	The timestamp (expressed as POSIX time).
hits	This row and nested fields are populated for all types of hits. Provides a record of all page visits.
customDimensions	This section contains any user-level or session-level custom dimensions that are set for a session. This is a repeated field and has an entry for each dimension that is set.
totals	This set of columns mostly includes high-level aggregate data.

Sélection et traitement des colonnes

Dans le cadre de notre étude, nous avons un dataset qui liste des transactions d'achat. Ce dataset comporte 903653 lignes et 55 colonnes.

Afin de réduire le nombre de variables, nous avons exploré et essayé de comprendre chaque colonne de notre dataset afin de juger leur importance.

Nous avons commencé par remplacer tous les "Nan" de notre target 'TransactionRevenue' par 0 et avons ainsi remarqué que 98.7% de notre target sont des valeurs nulles.

Ensuite nous avons d'abord supprimer de notre dataset la colonne 'sessionId' qui est déjà contenue dans 'fullvisitorID' et 'visitorID', toutes les colonnes contenant qu'une seule valeur ainsi que 'date' qui est redondant avec 'visitStartTime'. Ce qui a permis d'en éliminer 22.

Puis nous avons remplacé les "Nan" de certaines colonnes booléennes par True ou False. Les colonnes concernées sont 'isTrueDirect', 'bounces', 'newVisits', 'isVideoAd'.

Afin d'avoir des données plus représentatives, nous avons en premier lieu pensé à supprimer les colonnes qui avaient plus de 95% de données inexploitables. Voici les colonnes en question :

```
# columns with their proportion of unavailable data
for col in df.columns:
    counts = df[col].value_counts()
    total = df[col].isna().sum()
    if 'not available in demo dataset' in counts:
        total += counts['not available in demo dataset']
    if '(not set)' in counts:
        total += counts['(not set)']
    if '(not provided)' in counts:
        total += counts['(not provided)']
    if total/len(df[col]) > 0.9:
        print(col, total/len(df[col]))
```

```
campaign 0.9576098347485152
keyword 0.961975448540535
adContent 0.9878869433289106
page 0.9762519462669852
slot 0.9762519462669852
gclId 0.9761401777009538
adNetworkType 0.9762519462669852
isVideoAd 0.9762519462669852
```

On a décidé de tout de même garder ces données car même si une colonne a uniquement 5% de données significatives, elles pourraient être utiles pour plus tard.

Applications du log sur les données :

On a appliqué le log.1P sur les 'transactionRevenue' afin d'avoir des valeurs plus petites et une répartition normale.

```
[ ]  # prendre log(1+x) du transactionRevenue
     df['transactionRevenueLog'] = np.log1p(df['transactionRevenue'])
```

Création de colonne :

Nous avons créé une colonne ‘cum_sum_revenue’ afin d’avoir le revenu cumulé d’un visiteur, en comptant toutes ses visites précédentes. Premièrement nous avons trié notre dataset de telle sorte à regrouper par ‘visitorID’. De ce fait, pour chaque ligne, ‘cum_sum_revenue’ correspond à la valeur cumulée des transactions du visiteur à sa n-ième visite.

Nous avons aussi créé les colonnes ‘total_hits’ et ‘time_since_first_visit’, qui sont de la même façon le nombre total de visites et le temps depuis la 1ère visite d’un client.

De plus, pour réduire le nombre de colonnes one hot, nous n’avons gardé que les « sources » qui apparaissent plus de 90 fois. Les autres sont regroupées dans une variable « autre_sources »

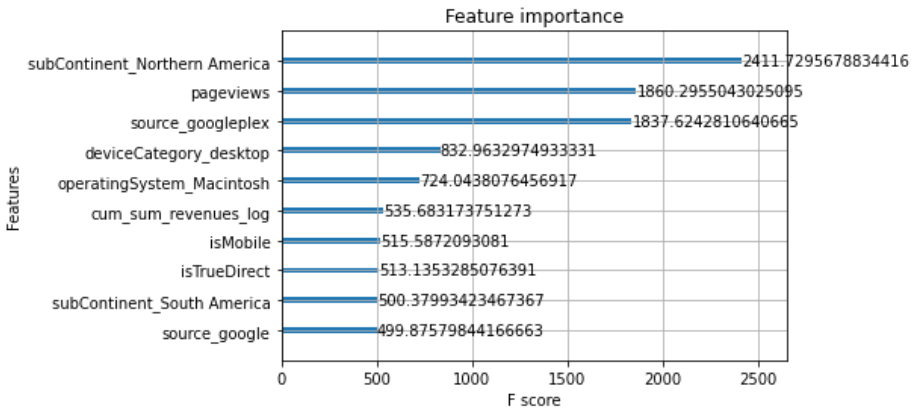
D’un autre côté, nous avons remarqué que seulement 2.5% des lignes ont la colonne ‘gclID’ de renseigné. Nous avons émis l’hypothèse que le fait de savoir que la ligne possède un ‘gclID’ renseigné est plus intéressant que l’id en lui-même. De ce fait, nous avons créé une nouvelle colonne ‘has_gcl’ qui a pour valeur 1 si ‘gclid’ est renseigné et 0 sinon.

Nous avons effectué les mêmes actions pour ‘keyword’ et ‘adContent’.

Ensuite nous avons divisé la colonne ‘visitStartTime’ en plusieurs colonnes : ‘visit_year’, ‘visit_month’, ‘visit_day’, ‘visit_weekday’ et ‘visit_hour’.

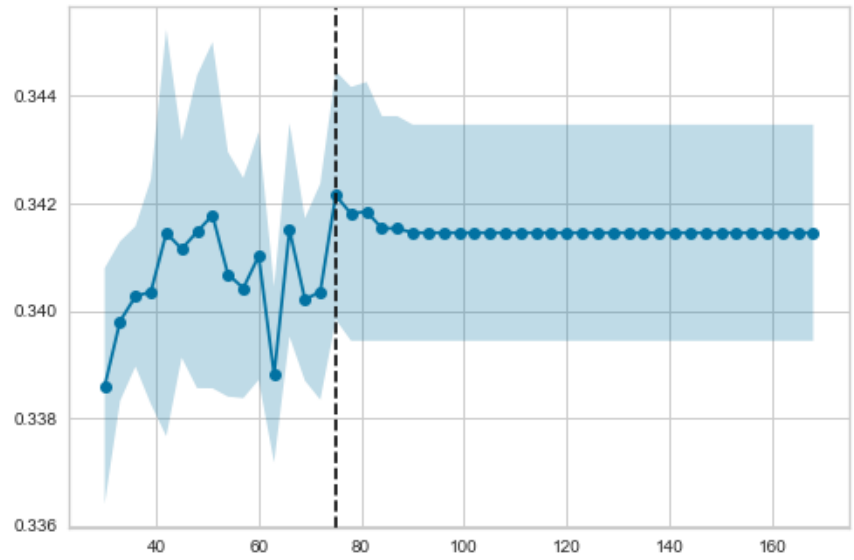
Visualisation de quelques features :

Voici les différentes features en fonctions de leurs importance pour XGBoost :



Également nous avons utilisés une fonction **Recursive feature elimination (RFE).** Cette fonction supprime les colonnes les moins importantes jusqu’à ce que le nombre spécifié de features optimal soit atteint. Ici nous obtenons un total de 75 features.

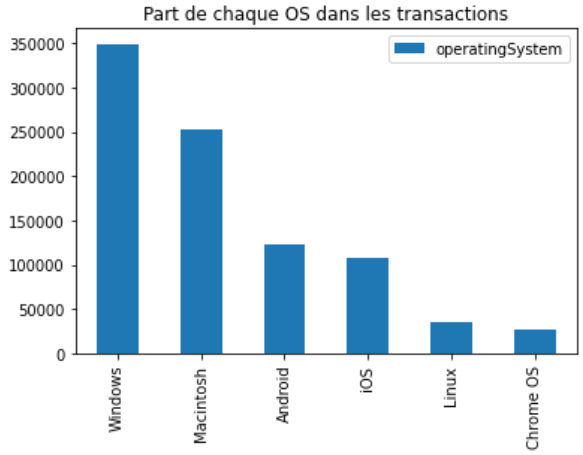
```
<matplotlib.lines.Line2D at 0x18ed19bc940>
```

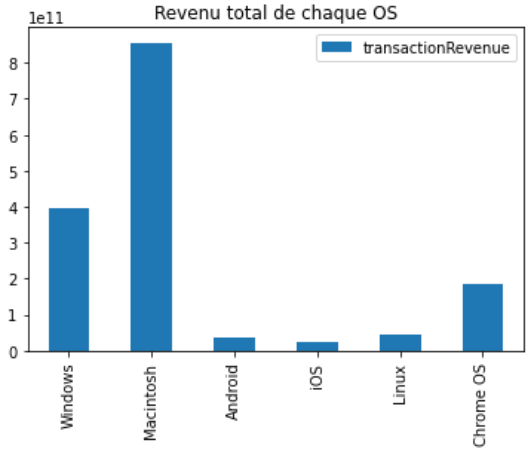


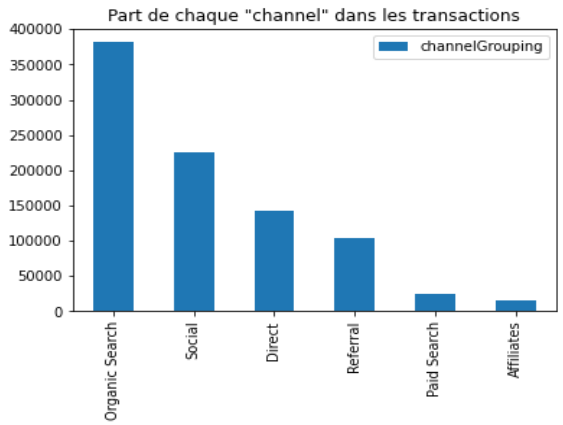

```
[ ]  f"Optimal number of features: {rfecv.n_features_}"
```

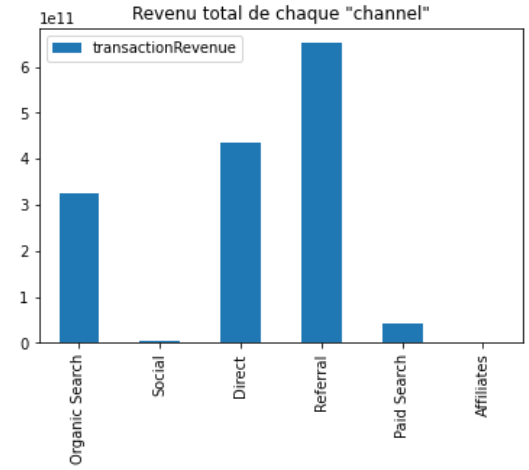
```
'Optimal number of features: 75'
```

Pour mieux visualiser notre target, nous l'avons représenté en fonction de quelques features:









On remarque que les OS majoritaires ne sont pas les mêmes que ceux qui constituent la majorité des revenues, et de même pour la variable « channel ». Globalement, les gens visitant les sites sur téléphone sont nombreux mais peu enclin à faire des achats. Les recherches directes rapportent le plus, avec celles des liens payants (referral).

Pour finir nous avons scalé nos données puis séparé notre dataset en train et en test set.

Nous avons essayé de séparer les données en fonction de leur date, mais nous obtenions des performances inférieures ainsi. Notre split est donc aléatoire, avec 80% de train set.

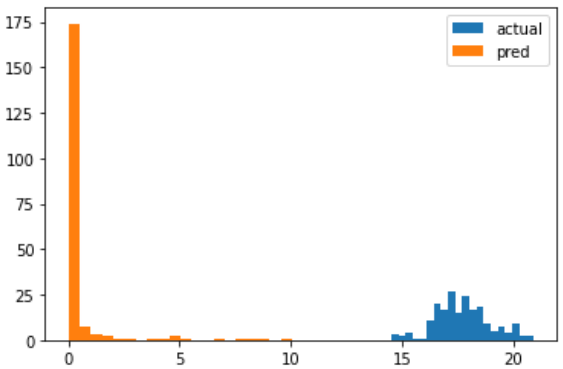
```python
def scale_data(train: pd.DataFrame, test: pd.DataFrame):
    # Scale the data
    scaled_cols = [
        "hits",
        "time_since_first_visit",
        "total_hits",
        "visitNumber",
        "visitStartTime",
        "visit_year",
        "visit_month",
        "visit_day",
        "visit_weekday",
        "visit_hour",
    ]
    # train sklearn StandardScaler on train set
    scaler = StandardScaler()
    scaler.fit(train[scaled_cols])
    # transform train and test sets
    train[scaled_cols] = scaler.transform(train[scaled_cols])
    test[scaled_cols] = scaler.transform(test[scaled_cols])
    return train, test


def get_X_y_train_test(train, test):
    X_train, y_train = (
        train.drop(["transactionRevenueLog"], axis=1),
        train.transactionRevenueLog,
    )
    X_test, y_test = (
        test.drop(["transactionRevenueLog"], axis=1),
        test.transactionRevenueLog,
    )
    return X_train, X_test, y_train, y_test
```

II. Description des approches utilisées

1ère approche : Régression sur les données

Pour notre première tentative, nous avons donc essayé plusieurs modèles de régression afin d'avoir une idée des performances que l'on pourrait obtenir. Nous avons pu remarquer que le modèle XGBoost avait les meilleures performances sans changer les hyperparamètres, et avons donc majoritairement utilisé celui-ci. Le détail des algorithmes utilisés est étudié la partie III.

Voici la répartition des prédictions où la **valeur réelle est strictement positive** pour ce modèle :



On se rend compte que la grande majorité des valeurs est proche de 0, et que nos prédictions ne sont donc pas vraiment intéressantes. Cela est probablement dû au fait que les classes ne soient pas du tout équilibrées, problème que nous étudierons dans la 3ème approche.

2ème approche : Classification des personnes n'ayant pas payé, puis régression sur les personnes ayant payé

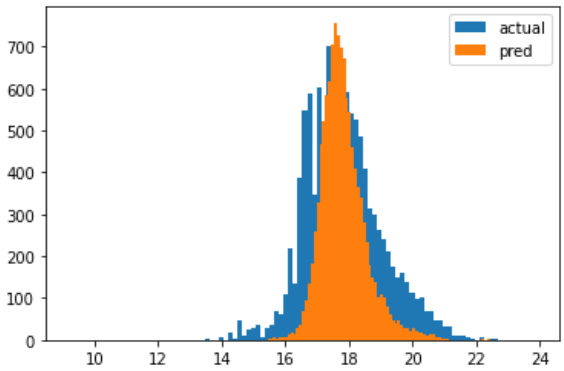
Etant donné la disparité entre les personnes ayant payé et les autres, nous avons eu l'idée de commencer par un modèle de classification afin de séparer les personnes ayant un transactionRevenue de 0 et ceux ayant effectué une transaction. Ensuite, nous avons entrainé un modèle de régression seulement sur les personnes ayant payé, et l'avons appliqué aux visiteurs ayant été classifiés comme ayant payé.

Pour ce modèle, nous avons aussi testé plusieurs algorithmes (XGBoost, RandomForest et DecisionTree). Nous avons évalué ces modèles en utilisant le recall_score, qui est bien adapté à nos classes mal réparties. Le modèle DecisionTree a obtenu le meilleur score avec environ 0.38 en utilisant comme critère « entropy » au lieu de « gini », ce qui signifie que l'on classifiait correctement 38% des visiteurs ayant payé.

```
1  print(classification_report(y_test_classif, classif.predict(X_test_classif)))
✓ 13s
              precision    recall  f1-score   support

       False       0.99      0.99      0.99    178094
        True       0.35      0.39      0.37      2280

    accuracy                           0.98    180374
   macro avg       0.67      0.69      0.68    180374
weighted avg       0.98      0.98      0.98    180374
```

```
1  classif = DecisionTreeClassifier(criterion="entropy")
2
3  classif.fit(X_train_classif, y_train_classif)
4  recall_score(y_test_classif, classif.predict(X_test_classif))
✓ 12.8s
0.38903508771929823
```

Nous avons donc entrainé le modèle de régression sur les lignes où le revenu était positif et l'avons appliqué à nos données. On obtient alors un score R^2 de 0.26 seulement, mais on observe que l'erreur RMSE est bien plus petite, avec seulement 1.03 environ contre 1.6 avec la première méthode.

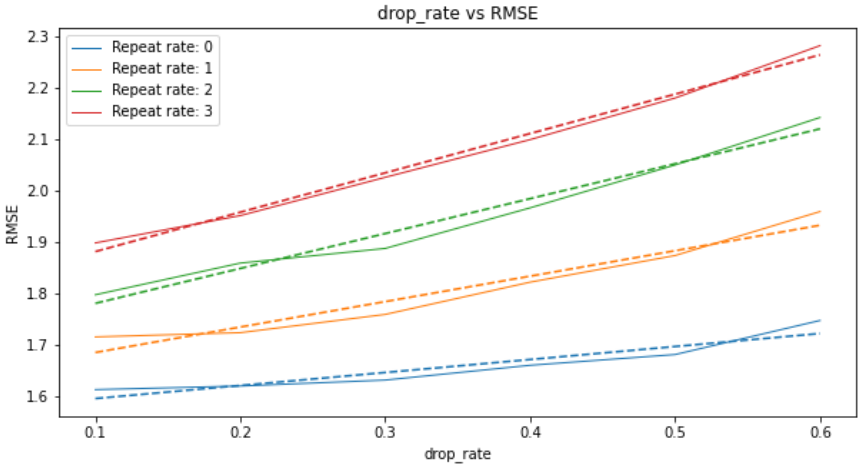


Voici la répartition obtenue dans notre 2ème méthode pour les **valeurs strictement positives de transactionRevenueLog**. On observe qu'elle est très similaire à celle des vraies données.

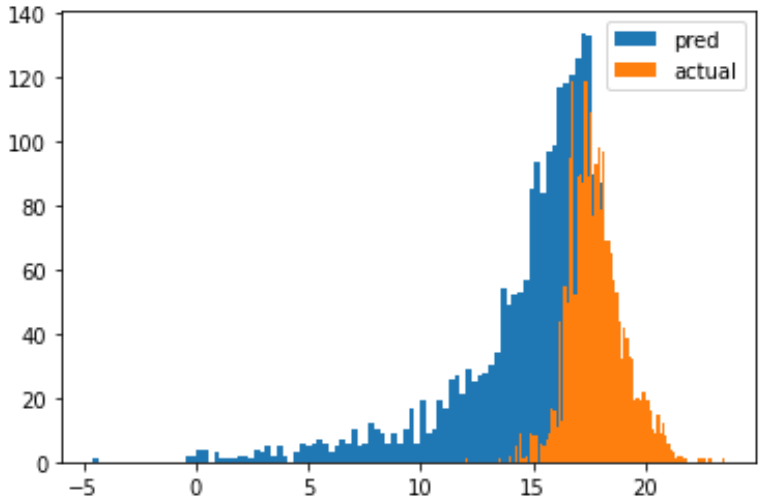
Pour conclure, cette méthode est plus efficace que la première pour prédire correctement le transactionRevenue des visiteurs. Cependant, comme expliqué plus tôt, moins de 40% des visiteurs ayant payé sont correctement classifiés, donc la majorité sont considérés comme n'ayant pas fait de transaction par notre modèle.

3ème approche : data augmentation

La dernière approche que nous avons testée a pour but de réduire la disparité entre les deux classes du dataset (personnes ayant effectué une transaction ou non). Pour cela, nous avons combiné deux méthodes : ignorer une partie des lignes où transactionRevenueLog valait 0, et dupliquer les lignes ou transactionRevenueLog était strictement positif. Ainsi, nous espérions réduire l'impact des revenus égaux à 0 dans notre modèle. Cependant, aucune de ces méthodes n'a permis de réduire notre R^2, et donc d'améliorer nos prédictions. Voici la RMSE obtenue pour plusieurs valeurs de drop_rate (pourcentage de lignes où le revenu vaut 0) et de repeat_rate (nombre de fois où l'on répète les lignes où le revenu est positif) :



Voici cependant la répartition des valeurs prédites quand la vraie valeur est **strictement positive**, avec un drop_rate à 0.5 et un repeat_rate à 10 (Environ 22% des valeurs de notre train et test set ont maintenant un transactionRevenue > 0) :



Une fois encore, la répartition semble mieux que pour la première approche mais moins précise comparé à la seconde approche.

III. Les algorithmes testés, leurs résultats, leurs performances, et les variables

A. Algorithmes utilisés

Voici les algorithmes que l'on a testés afin de savoir quels étaient leur performance :

- **Random Forest Regresso**
- **Lasso**
- **DecisionTreeRegressor**
- **XGBRegressor**

Les résultats :

```
RandomForestRegressor
Lasso
DecisionTreeRegressor
XGBRegressor
```

	RandomForestRegressor	Lasso	DecisionTreeRegressor	XGBRegressor
R^2	0.2913	0.154	-0.3588	0.3378

B. Algorithme le plus performant

Après avoir testé les algorithmes au-dessus, **XGBRegressor** a donné le meilleur résultat c'est donc ce modèle qu'on utilisera pour prédire notre valeur.

Voici les variables utilisées :

```
subsample=0.9

reg_lambda=1

reg_alpha=2

learning_rate=0.1

n_estimators=180

tree_method="hist"
```

De plus, voici les prédictions de regression pour y_test >0, **y_test** étant la valeur que nous voulons prédire et **pred** notre prédiction.

Malgré un R^2 élevé nous obtenons que des 0 en prédiction. Ces résultats sont expliqués par le fait que le dataset ne soit pas assez bien reparti.

	y_test	pred
525168	17.229624	0.000000
300616	16.212496	0.000000
339007	17.540481	0.000000
309219	17.407604	0.000000
721891	17.840684	0.000000
197716	18.197412	0.000000
665240	17.516072	0.000000
258680	18.287149	0.000000
434918	17.033986	0.000000
584809	17.840505	0.000000
472484	18.420581	0.000000
726126	17.567834	0.000000
63931	14.910784	0.007626
572837	17.225998	0.000000
524129	17.521493	0.000000
755013	20.296352	0.000000
313930	18.023035	0.000000
57155	16.923678	-0.001958
174861	17.973299	0.000934
296275	18.155151	0.000000

IV. Meilleure évaluation des prévisions

Pour conclure, l'approche la plus concluante semble être de classifier dans un premier temps les personnes qui auront un revenu positif des personnes avec un revenu égal à 0, puis d'appliquer un modèle de régression seulement sur les personnes classifiées comme ayant dépensé de l'argent. En effet, bien que seulement 40% des clients ayant payé sont correctement classifiés, la répartition des revenus prédits est bien meilleure avec cette approche