PROJET : Kaggle *Rossmann Store Sales 2015*

ENSIIE et Université d'Évry - Université Paris-Saclay

Apprentissage statistique - Modélisation Statistique

ETUDIANTS

Abdoulaye SAKHO, Lilian LECLERC, Samuel SABO & Adrien BOLLING
Groupe 20

PROFESSEURS

V. Bourgeais, B. Hanczar & J.-C. Janodet

Mars 2022

Table des matières

Introduction

Ce projet d'apprentissage automatique se base sur les données d'une compétition de prédiction Kaggle, et plus exactement sur la compétition Rossmann Store Sales de 2015. Nous avons choisi de le traiter en Python

Rossmann possède en effet plus de 3000 pharmacies dans 7 pays européens. Lors de la mise en place de ce concours, les directeurs des différents magasins Rossmann étaient chargés de prévoir leurs ventes quotidiennes jusqu'à 6 semaines en amont.

Cependant, les ventes en magasin à prédire sont influencées par de nombreux facteurs, tels que les diverses promotions, la concurrence, les vacances scolaires et jours fériés, la saisonnalité, la localité, etc. De plus, une prédiction individuelle des divers magasins peut conduire à des résultats très variables, souvent erronés, une fois généralisée à l'ensemble des magasins Rossmann.

Ainsi, le but final de cette compétition, et donc de notre projet, est de mettre en place une stratégie prédictive en se basant sur l'ensemble des magasins Rossman sur une période donnée, et de s'approcher au maximum des ventes réelles effectuées sur la période de 6 semaines s'étalant sur les mois d'août et septembre 2015.

1 Traitement de la base de données

1.1 Présentation

Les données fournies sont les données historiques relatives à 1115 magasins Rossmann. Nous devons, à partir de ces données, prédire les ventes des magasins présents dans la base de données test. De plus, certains magasins peuvent être temporairement fermés pour cause de rénovation.

Les données fournies sont réparties de la sorte, il existe 3 fichiers train.csv, test.csv et store.csv qui contiennent des données correspondantes aux divers magasins Rossman, et un fichier supplémentaire sample_submission.csv permettant de soumettre les résultats de nos diverses stratégies directement sur Kaggle.

Plus exactement, train.csv et test.csv contiennent des informations historiques liées directement aux ventes des magasins Rossman, tandis que store.csv contient davantage de renseignements supplémentaires, notamment contextuels, sur ces différents magasins.

Les différentes variables présentes dans test.csv et train.csv sont les suivantes :

- Store : un ID unique propre à chaque magasin
- DayOfWeek : le jour de la semaine (numéroté de 1 à 7)
- Date : la date du jour (format YYYY-MM-DD)
- Sales : uniquement présente dans test.csv (en effet, c'est la variable à prédire dans train.csv), représentant le nombre de ventes effectuées par le couple (Store, Date)
- Customers : le nombre de clients dans un magasin à un jour donné
- Open : un indicateur permettant de savoir si le magasin est ouvert (1) ou fermé (0)
- Promo : indique si un magasin effectue ou non des promotions ce jour
- StateHoliday : indique si des vacances nationales ont lieu ou non (0 ou 1). En général tous les magasins, hormis quelques exceptions, sont fermés pendant ces vacances. De plus, toutes les écoles sont fermées lors des jours fériés et des week-ends. On suit la notation suivante : jour férié (a), vacances de Pâques (b), vacances de Noël (c), pas de vacances (0)
- SchoolHoliday : indique si le couple (Store, Date) est affecté ou non par la fermeture des écoles publiques

Ensuite, voici les différentes variables de store.csv :

- Store : une ID unique propre à chaque magasin
- StoreType : différencie 4 types de magasins (a, b, c ou d)
- Assortment : décrit un niveau d'assortiment (basique (a), supplémentaire (b), étendu (c))
- CompetitionDistance : distance en mètres par rapport au concurrent le plus proche
- CompetitionOpenSince[Month/Year] : donne l'année et le mois approximatifs depuis lequel le premier concurrent a été ouvert
- Promo2 : Promo2 est une promotion continue et consécutive pour certains magasins, si le magasin participe (1) sinon (0)
- Promo2Since[Year/Week] : indique l'année et la semaine calendaire correspond au début de la participation à cette promotion
- PromoInterval : décrit les intervalles consécutifs pour lesquels Promo2 se déroulent, en nommant les mois durant lesquels la promotion est relancée. Par exemple, "Feb,May,Aug,Nov" signifie que pour chaque année donnée, Promotion2 commence à ces périodes

1.2 Restructuration primaire des données

Dans cette section nous allons parler du premier traitement auquel nous avons soumis les données. En effet, dans train.csv et store.csv, il existe des données manquantes et certaines variables sont qualitatives. Les modèles de Machine Learning que nous avons utilisé au cours de ce projet ne fonctionnent pas avec des données manquantes ni avec des variables qualitatives. C'est pour cette raison qu'il faut modifier nos données avant d'essayer un modèle.

Il est donc clair que la restructuration des données impacte les modèles et donc par extension, les résultats/les scores, obtenus à partir de ces modèles. C'est pourquoi nous avons, à plusieurs reprises, modifié nos

données pour tenter d'améliorer nos modèles au cas par cas. Ici nous présenterons uniquement la restructuration primaire (la première que nous avons effectuée sur nos données), qui nous sert de base départ.
Nous avons commencé par fusionner les fichiers store.csv et train.csv, et nous avons ensuite supprimé la variable *Customers*, car elle n'est pas présente dans test.csv. 2mm Une représentation de la matrice de corrélation de nos données (*cf Figure 5*), nous montre qu'aucune variable n'est redondante. (car nous avons déjà retiré Customers).

En ce qui concerne les variables qualitatives, nous les avons transformées en variables quantitatives par simple correspondance. Par exemple, la variable *StoreType* qui prend à l'origine les valeurs (a,b,c,d), prend après restructuration les valeurs (0,1,2,3). Nous sommes conscients qu'une telle transformation altère les modèles dans le sens où certains modèles peuvent penser que plus la valeur qualitative prise par la variable est grande, plus la corrélation entre celle-ci et la variable cible est grande. Une alternative aurait été de faire 4 nouvelles variables binaires qui indique chacune l'appartenance un type de boutique donné. Ce schéma a été suivi pour les variables *StoreType* et *Assortment*.

En ce qui concerne les variables manquantes, nous avons remplacé les `NaN` par la moyenne de chaque variable, en l'occurrence les variables *CompetitionDistance* et *CompetitionOpenSince[Month/Year]*.

Les variables *Promo2Since[Year/Week]* ont été supprimées. Les variables *Promo2* et *PromoInterval* ont été fusionnées pour donner naissance à une variable Promo qui vaut 0 si la boutique en question ne fait pas de promotions, et 1 si la boutique en fait sur l'intervalle "March,Jun,Sept,Dec" et ainsi de suite.

La variable date a été séparée en 3 variables, une pour le jour, une pour le mois et une dernière pour l'année.

L'aspect final de nos données après la restructuration primaire est donnée en *Annexe 1*. Nous avons utilisé la fonction .info() du module Pandas de Python. L'annexe *CODE 1* correspond aux commandes utilisées en Python afin d'effectuer toutes les modifications présentées précédemment.

2 Régression à l'aide des variables explicatives

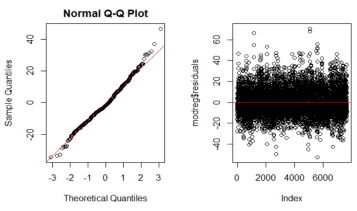
Cette section est la première concernant la construction de modèles. Ici, nous allons utiliser toutes les variables explicatives à notre disposition afin de construire des modèles.

2.1 Les modèles linéaires par la méthode des moindres carrées ordinaires (MCO)

La méthode des MCO a pour but la minimisation de la quantité $E(\beta)$ selon l'argument β.En résolvant ce problème d'optimisation, la méthode MCO nous donne une expression algébrique de $\hat{\beta}$ en fonction de Y et X. Mais ce résultat n'est valable que si la matrice X^TX **est inversible !**

De plus,en supposant que **les résidus sont indépendants et identiquement distribués de cette façon :** $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, on peut retrouver la loi de tous nos estimateurs ($\hat{\beta}$ et $\hat{Y}$). Avec la loi de $\hat{\beta}$, on parvient à construire un test de significativité pour chaque coefficient, c'est le **test de Student**. Ce test nous permet de savoir si une des variables explicatives du modèle, à travers la valeur de son coefficient donné, est significative ou non (c'est-à-dire si on peut forcer la valeur de ce coefficient à valoir 0 dans le cas où la variable concernée est non significative). Toutefois, ce test n'a de sens que si **les variables sont peu corrélées. !**.

Dans notre cas, la matrice X^TX est inversible. De plus, la gaussianité et l'homoscédasticité des résidus sont vérifiées. En effet sur la *Figure ci-contre*, on observe sur le premier graphique que les quantiles de nos résidus suivent ceux d'une loi normale, l'hypothèse de normalité des résidus est vérifiée. Sur le second graphique, on observe une répartition uniforme des résidus de part et d'autre de la droite d'équation y=0,



l'hypothèse d'homoscédasticité des résidus est vérifiée. Enfin, nous avons démontré au **Rendu2** que nos données compressées sont peu corrélées.

Toutes les hypothèses du modèle linéaire étant vérifiées, nous pouvons utiliser des MCO pour nous fournir un estimateur optimal.

Résultats :

Les modèles linéaires ont été nos modèles de départ pour ce projet. Avec la restructuration primaire des données nous avons obtenu un score sur Kaggle de 0.66. Après de multiples modifications des données, en rendant par exemple les données temporelles cycliques, la modélisation linéaire n'obtenait jamais de très bons scores. Il s'agit donc d'un modèle trop simpliste dans notre cas.

2.2 Les modèles de pénalisation

Une fois le modèle linéaire établi, un calcul de la performance de nos modèles nous permet d'évaluer leur performance et leur optimalité. On peut notamment voir si nous ne sommes pas victimes **d'over-fitting**, ce qui peut être problématique et surtout ne pas générer de bonnes performances une fois notre modèle évalué sur d'autres sets de données. C'est alors qu'interviennent les méthodes pénalisées, et notamment les méthodes de **Ridge** et **Lasso**.
Avant d'aller plus loin, rappelons la forme que prend une modélisation linéaire : $Y = X\beta + \epsilon$ où ϵ est le biais du modèle, c'est à dire, la différence entre l'espérance de notre estimateur et sa valeur théorique attendue. Les méthodes de pénalisation consistent à introduire une pénalité permettant d'affiner notre modèle en réduisant la variabilité de celui-ci.
Pour la méthode de Ridge, cette pénalité sera de la forme $\lambda \sum_{i \leq p} \beta_i^2$, λ étant le coefficient de pénalité permettant de contrôler son impact, il peut être fixé, mais pourra être déterminé, si ce n'est affiné, durant le processing du nouveau modèle.
Pour la méthode de Lasso, la forme de la pénalité diffère seulement par le choix de la norme, en effet, elle s'écrit $\lambda \sum_{i \leq p} \mid \beta_i \mid$ et on reconnaît la norme 1.

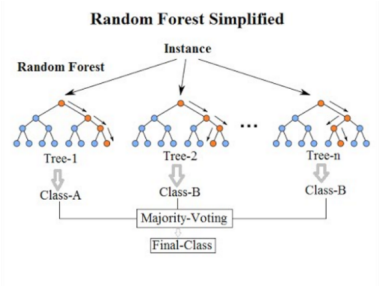
Hyperparamètres et résultats :

Les modélisations Ridge et Lasso sont plus complexes qu'une simple modélisation linéaire. Un hyperparamètre α est à choisir pour ces modèles, et cela est effectué par des fonctions de Python prédéfinies. Par exemple, pour la régression Lasso, nous avons utilisé la fonction LassoCV de la librairie ScikitLearn de Python. Cette fonction choisit le meilleur hyperparamètre λ, selon un critère de Cross-Validation, parmi une plage de valeurs qui lui ont été fournies (*cf CODE 2*).

Cependant, ces modèles ont été décevants, nous offrant dans le meilleur des cas des scores de 0.7 sur Kaggle (avec Lasso). Nous avons donc tenté d'améliorer ces modèles en effectuant une régression Lasso par type de boutique. *L'Annexe 1* illustre la répartition des 4 types des différentes boutiques. En supposant que des boutiques du même type ont un comportement similaire, nous pensions qu'effectuer une régression par boutique améliorerait notre score. Ce qui ne fut pas le cas.

2.3 Les modèles de Forêts aléatoires

La modélisation par forêts aléatoires (ou Random Forest en anglais) est un algorithme classique de l'apprentissage automatique, initialement utilisé pour la classification, se fondant sur le croisement de plusieurs arbres de décisions de profondeurs identiques.
Présentons d'abord ce qu'est un arbre de décision, et comment il est utilisé dans le cadre d'un Random Forest. Comme l'indique son nom, un arbre de décision est un outil de décision représentant un ensemble de questions et de réponses permettant de classifier un élément/échantillon dans un ensemble plus grand. Chaque noeud de l'arbre est une question, dont les branches sont les différentes réponses possibles à ladite question. Dans les cas les plus généraux, l'arbre est binaire, c'est-à-dire que les uniques réponses possibles aux questions sont oui et non, et les feuilles signifient l'adhésion ou non de l'élément/échantillon à un ensemble.
On peut toutefois adapter le problème à des problèmes de régression, et non seulement de classification. On peut par exemple définir des paliers plutôt que de se limiter à une appartenance ou non à un seul ensemble.



Toutefois, des packages sont disponibles pour plusieurs langages de programmation, permettant d'éviter à avoir à

1. trouver les questions à poser et la décision à prendre en fonction de la question
2. définir une stratégie d'adaptation de l'algorithme de classification à un objectif de régression.

Nous utiliserons la fonction RandomForestRegressor du package Scikit-Learn disponible sur Python, pour effectuer directement les régressions souhaitées.

Hyperparamètres et résultats : Nous avons utilisé les hyperparamètres par défaut de la fonction RandomForestRegressor de Scikit-kLearn. Nous avons modifié le paramètre random_state en le fixant à 2, mais cela n'a pas amélioré nos scores (*CODE 3*). Un première régression Random Forest sur les données de la restriction primaire des données nous a permis d'obtenir un score de 0.16 sur Kaggle. en modifiant les scores, nous avons atteint les 0.1400.

Comme précédemment nous avons effectuer une régression de RandomForest pour chaque type de boutique. Cela nous a donné des scores de 0.18. Finalement le meilleur score obtenu avec un Random Forest est celui sur la restructuration primaire des données, en utilisant les paramètres par défaut de la fonction RandomForestRegressor de Scikit-Learn. Ce score publique vaut : 0.1400.

3 Séries temporelles

Les séries temporelles (ou séries chronologiques) sont des séries d'observations d'une (ou plusieurs) quantité(s) au cours du temps. Dans la suite de notre partie de modélisation, c'est le point de vue que nous allons adopter.

Dans le cas de notre projet, nous allons maintenant "abandonner" toutes les variables explicatives. Notre variable cible "Store", le chiffre d'affaire journalier, sera notre seule variable explicative et toujours notre variable cible.
Nous aurons donc au total 1115 séries chronologiques à étudier (en réalité on en aura moins car les données de tests ne concernent pas les 1115 boutiques Rossmann).

3.1 Les modèles de moyennes mobiles MA(q)

Dans un modèle de moyenne mobile, on suppose que notre variable cible à un instant t s'écrit comme une combinaison linéaire finie des valeurs de la série correspondant à des dates entourant t :

$$MA_h(t) = \sum_{j=-h}^{h} \alpha_j \, X_{t+j}$$

Il existe de nombreux modèles de moyennes mobiles. Ici, nous allons utiliser le modèle plus simple, celui de la moyenne mobile aléatoire d'ordre q (q $\in N$) qui après transformation peut s'écrire sous la forme suivante :

$$MA(q) : X_t = \mu + \theta_1\omega_{t-1} + \theta_2\omega_{t-2}... + \theta_q\omega_{t-q} \quad avec\, \mu = E[X_t]\ , et\ \omega_i \sim \mathcal{N}(0,\, \sigma^2)$$

Les ω_i sont les termes d'erreur de bruit (bruit blanc).

La discrimination des modèles MA(q) (pour différentes valeurs de q) a été effectué à l'aide du calcul de la Racine-carré de la Moyenne du Pourcentage d'Erreur (**RMPE**, aussi appelé RSME) sur les données de validation.

Résultats :

Les modèles MA(q) ont servi de base départ pour les modèles de séries chronologiques. En effet, les modèles MA(q) nous ont permis d'obtenir dans le meilleur des cas des scores de 0.65. Ce fut notre base de départ.

3.2 Les modèles Autoregressifs AR(p)

Un processus autorégressif est un modèle de régression pour séries temporelles dans lequel la série est expliquée par ses valeurs passées plutôt que par d'autres variables.

$$X_t = \sum_{k=1}^{p} a_k X_{t-k} + \varepsilon_t \quad \varepsilon_t \sim \mathcal{N}(\mu, \sigma^2)$$

$\varepsilon_1, \ldots, \varepsilon_T$ indépendantes

Toutefois, ce type de modèle ne peut être utilisé qu'après s'être assuré que la série étudiée est **stationnaire** !

Hypothèses et Hyperparamètres : La discrimination des modèles AR(p) (pour différentes valeurs de p) a été effectué à l'aide du calcul de la Racine-carré de la Moyenne du Pourcentage d'Erreur (RMPE) sur les données de validations (cf *CODE 4*).

La stationnarité des 1115 séries a été vérifiée en les séparant par types de boutique et en regardant de nombreux représentants de chaque classe. L'annexe 3 illustre un exemple.

Résultats :

Les modèles AR(p) ont commencé par nous donner des scores assez décevant de l'ordre de 0.40. le termes "décevant" est utilisé ici car les modèles AR(P) nous donnait de très bons scores sur les données de validation (*cf Annexe 4*).

En modifiant les données d'entraînement de ces modèles, nous avons significativement améliorer leurs performances. En nous entraînant sur les données d'avril à juillet 2014, en validant nos hyperparamètres sur les données de août et septembre, nous avons obtenu un score privé de 0.24 avec les modèles AR(p).

Conclusion

En conclusion, ce projet d'apprentissage fut une bonne initiation à la plateforme Kaggle. Nous avons ainsi pu observer, à travers un exemple concret, comment peuvent être utilisé les outils de Machine Learning dans le domaine de l'indsutrie ou bien le secteur commercial. L'étude des Rossman stores fut aussi une belle occasion pour nous de mettre en oeuvre les connaissances que nous avons acquis au cours de notre cursus.

Après de multiples efforts, le meilleur score que nous avons obtenu vaut 0.1400 à l'aide d'un modèle par défaut de Random Forest. En essayant de complexifier ce modèle nous n'avons pas obtenu de meilleurs résultats. Notons donc que ce ne sont pas les modèles les plus complexes qui fournissent les meilleurs résultats.

Enfin, nous avons observé sur Kaggle que les équipes ayant obtenus les meilleurs scores ont recours à du boosting (le plus souvent XGBoost) et à des modèles complexe tel que le Facebook Prophet. Le boosting est donc une des solutions pour améliorer notre score final, mais son utilisation est prohibée lors de tels projets ce qui est logique.
Une autre piste aurait été

Annexe

```
Entrée [18]:  1  train2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1017209 entries, 0 to 1017208
Data columns (total 16 columns):
 #   Column                     Non-Null Count    Dtype
---  ------                     --------------    -----
 0   Store                      1017209 non-null  int64
 1   DayOfWeek                  1017209 non-null  int64
 2   Sales                      1017209 non-null  int64
 3   Open                       1017209 non-null  int64
 4   Promo                      1017209 non-null  int64
 5   StateHoliday               1017209 non-null  int64
 6   SchoolHoliday              1017209 non-null  int64
 7   StoreType                  1017209 non-null  int64
 8   Assortment                 1017209 non-null  int64
 9   CompetitionDistance        1017209 non-null  float64
 10  CompetitionOpenSinceMonth  1017209 non-null  float64
 11  CompetitionOpenSinceYear   1017209 non-null  float64
 12  Promo                      1017209 non-null  int64
 13  Year                       1017209 non-null  int64
 14  Month                      1017209 non-null  int64
 15  Day                        1017209 non-null  int64
dtypes: float64(3), int64(13)
memory usage: 131.9 MB
```

FIGURE 1 – Informations sur les données de la restructuration primaire

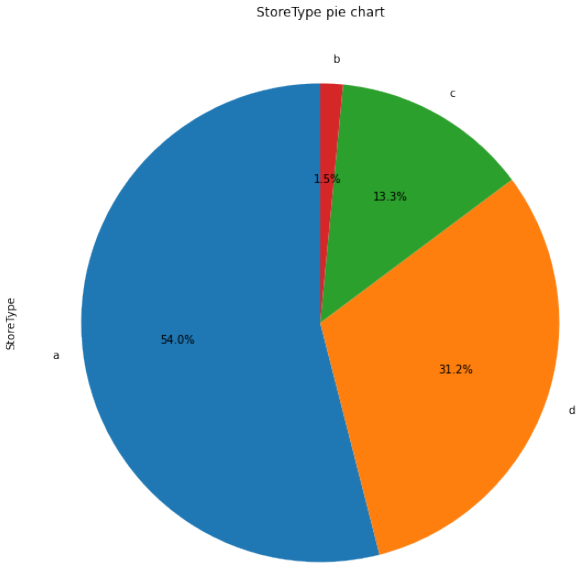


FIGURE 2 – Pie-chart des type de boutique

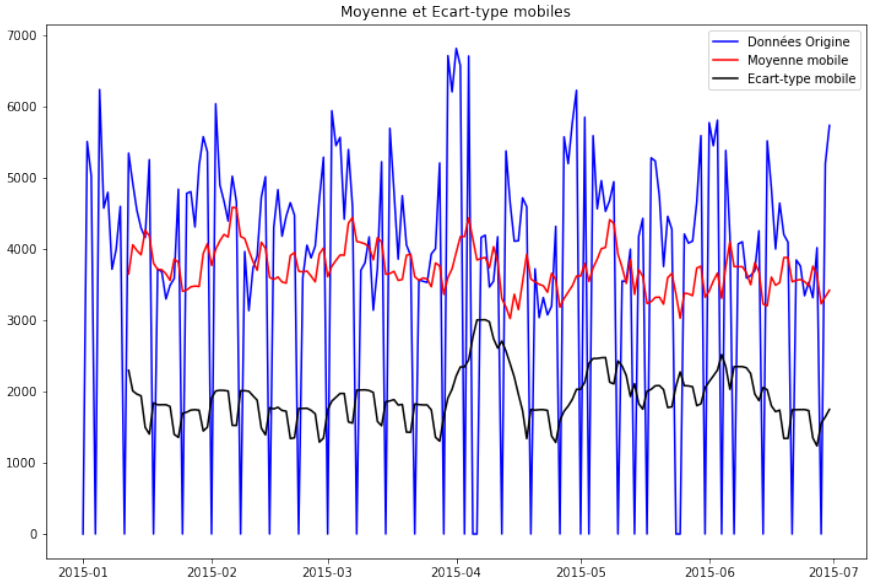
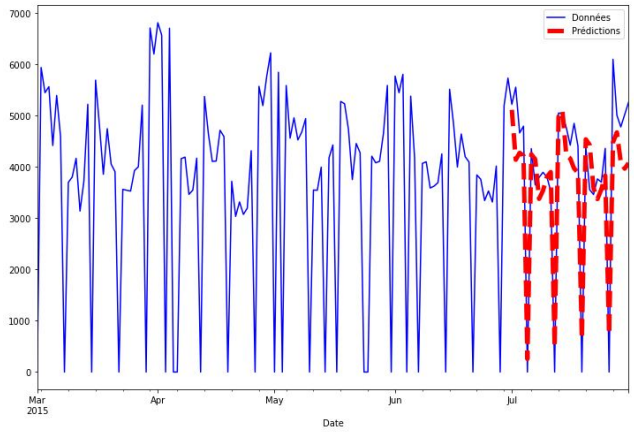


FIGURE 3 – Affichage stationnarité d'une série

Store type c:

Entrée [26]:

```
model1, rcarré, rmse, rmspe = AR_fonc(data=sales_c, n=50, start_train='2015-03-01',  end_train='2015-06-30',
                                      start_test='2015-07-01', end_test='2015-07-31')
print(r'$R^2$ vaut:',rcarré)
print('Le RMSE vaut:',rmse)
print('Le RMSPE vaut:',rmspe)
```




```
$R^2$ vaut: 0.8444927277763012
Le RMSE vaut: 644.3908329753119
Le RMSPE vaut: 0.13354374726823134
```

FIGURE 4 – Train et Validation d'une seule série avec un modèle AR(p)

CODE

Listing 1 – CODE 1 : Code source de la restructuration des données

```
1
2  ##########Onsupprime les colonnes dont on ne veut pas se servir#########
3  train2=train_store.drop(['Customers','Promo2SinceWeek','Promo2SinceYear'],axis=1)
4
5  ########################################On fill les na#####################
6  train2['CompetitionDistance'].fillna(value=train2['CompetitionDistance'].mean(), inplace=True)
7  train2['CompetitionOpenSinceMonth'].fillna(value=train2['CompetitionOpenSinceMonth'].mean(),
       inplace=True)
8  train2['CompetitionOpenSinceYear'].fillna(value=train2['CompetitionOpenSinceYear'].mean(),inplace=
       True)
9
10 #train2.fillna(value={'CompetitionDistance': train2[['CompetitionDistance']].mean(), '
       CompetitionOpenSinceMonth': train2[['CompetitionOpenSinceMonth']].mean(), '
       CompetitionOpenSinceYear': train2[['CompetitionOpenSinceYear']].mean()},inplace=True)
11
12 Promo = train2['PromoInterval'].map({ float('nan'): 0, 'Jan,Apr,Jul,Oct': 1,
13                                                    'Feb,May,Aug,Nov': 2, 'Mar,Jun,Sept,
       Dec': 3})
14 #Promo
15 train2[['Promo2']]=Promo
16 train2.drop(['PromoInterval'],axis=1,inplace=True)
17 train2.rename(columns={'Promo2':'Promo'}, inplace=True)
18
19 ########## On transforme la Data ################
20 train2['Year']=pd.DatetimeIndex(train2.Date).year
21 train2['Month']=pd.DatetimeIndex(train2.Date).month
22 train2['Day']=pd.DatetimeIndex(train2.Date).day
23
24 train2.drop(['Date'],axis=1,inplace=True)
25
26 #######On tranformes variables qualitatives en variables quantitatives#########
27 train2['StateHoliday'] = train2['StateHoliday'].map({'0': 0, 'a': 1, 'b': 2, 'c': 3})
28 train2['StoreType'] = train2['StoreType'].map({'a': 0, 'b': 1, 'c': 2, 'd': 3})
29 train2['Assortment'] = train2['Assortment'].map({'a': 0, 'b': 1, 'c': 2})
```

Listing 2 – CODE 2 : Code source du modèle Lasso simple

```
1
2  X_train = pd.concat([ K10fold[0].drop(['Sales'],axis=1), K10fold[1].drop(['Sales'],axis=1),
       K10fold[2].drop(['Sales'],axis=1)
3                      , K10fold[3].drop(['Sales'],axis=1), K10fold[4].drop(['Sales'],axis=1), K10fold
       [5].drop(['Sales'],axis=1),
4                      K10fold[6].drop(['Sales'],axis=1), K10fold[7].drop(['Sales'],axis=1)])
5
6  Y_train=pd.concat([ K10fold[0][['Sales']], K10fold[1][['Sales']], K10fold[2][['Sales']], K10fold
       [3][['Sales']],
7                      K10fold[4][['Sales']], K10fold[5][['Sales']], K10fold[6][['Sales']], K10fold
       [7][['Sales']] ])
8
9  X_test = pd.concat([ K10fold[9].drop(['Sales'],axis=1), K10fold[8].drop(['Sales'],axis=1) ] )
10 Y_test = pd.concat([ K10fold[9][['Sales']], K10fold[8][['Sales']] ] )
11
12 cv = RepeatedKFold(n_splits=10, n_repeats=3)
13
14 lcv_V2 = LassoCV(alphas=arange(0.01, 1, 0.01),cv=10)
15 lcv_V2.fit(X_train,Y_train)
16
17 Y_pred = lcv_V2.predict(X_test)
18
19
```

Listing 3 – CODE 3 : Code source du Random Forest simple

```
    from sklearn.model_selection import train_test_split

#donees_train=train2.drop('Sales',axis=1).loc[train2['Year']== 2014]
#donnees_test= train2.loc[train2['Year']== 2014][['Sales']]

donees_train=train2.drop('Sales',axis=1)[ (train2['Year']== 2014)  & (train2['Month'] >= 8 ) & (train2['Month'] <= 9 )]
donnees_test=train2[ (train2['Year']== 2014)  & (train2['Month'] >= 8 ) & (train2['Month'] <= 9 )][['Sales']]
X_train, X_test, y_train, y_test = train_test_split(donees_train, donnees_test, test_size=0.25)

clf=RandomForestRegressor()#random_state=2
clf.fit(X_train,y_train)

Y_pred=clf.predict(X_test)

clf.score(X_train,y_train)


```

Listing 4 – CODE 4 : Code source du calcul final des modèles AR(p)

```
def AR_final(data,data_comparaison, indices_alt, n, start_train, end_train, start_test, end_test):
    # n est la valeur max de p
    #data corresponds à notre 1115-Fold
    ####"indices_alt sont les indices pris en compte
    ####Attention": au max va de 0 à 1114
    #c'est une liste

    from statsmodels.tsa.ar_model import AutoReg
    from sklearn import metrics
    from sklearn.metrics import r2_score

    res=data[0].reset_index().iloc[[0]]

    for f in indices_alt: # boucle des indices qui voint être pris en compte

        X_train=Fold_Stores[f-1][start_train:end_train].asfreq('d')
        X_test=Fold_Stores[f-1][start_test:end_test]#.asfreq('d')
    ######################### regression ################################
        val_score = []
        for k in range(10, (n+1)):
            score1 = kaggle_error_vrai(AutoReg(X_train, lags=k,
                                               old_names=True).fit().predict(
                start=pd.to_datetime(start_test), end=pd.to_datetime(end_test), dynamic=False), X_test)
            val_score.append(score1)
        min_value = min(val_score)
        min_index = val_score.index(min_value)

        model = AutoReg( Fold_Stores[f-1]['2015-04-01':'2015-07-31'].asfreq('d') , lags= (min_index+10),old_names=True)# k entier et index commence à 0
        model_fit = model.fit()#.fit(disp=0) pour cacher (ou -1)
        X_pred = model_fit.predict(start=pd.to_datetime('2015-08-01'), end=pd.to_datetime('2015-09-17'), dynamic=False)

    ################################  résultat final#################################
        X_pred=X_pred.reset_index()
        X_pred.rename(columns = {'index' : 'Date', 0 : 'Sales'}, inplace = True)
        res=pd.concat([res,X_pred],ignore_index=True,axis=0)

    res=res.drop(0)
    return res

```

Table des figures

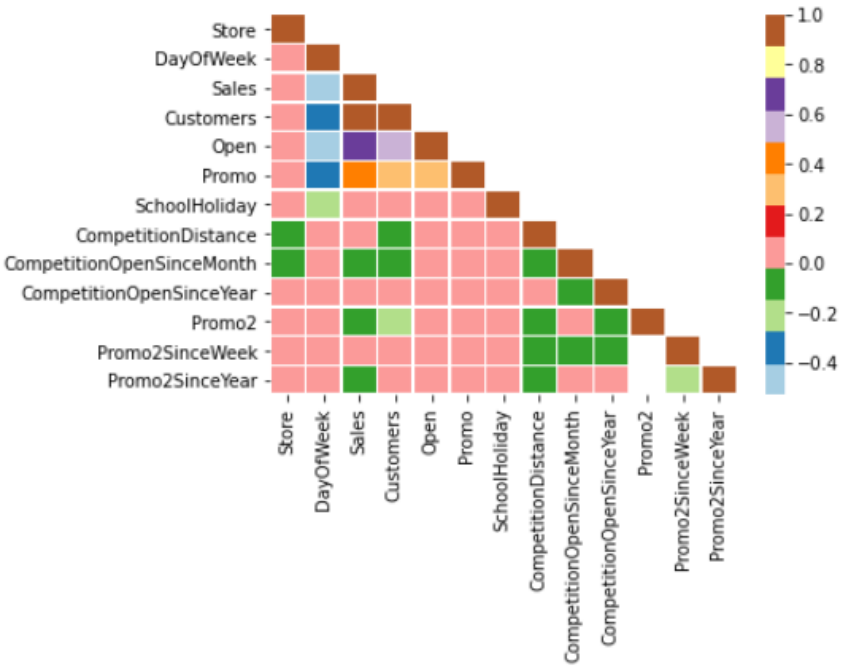


FIGURE 5 – Tableau de corrélation des données