TROUDE Jean-Benoit Dimanche 9 avril 2021

Mineur Recherche

Manuel Utilisateur

-

Sujet

Forecast of banks 'default and firms ' returns using their respective historical financial statements

Sommaire

Description brève des fichiers

- Fichier2.2_parametres : comparaison capital_ratio_1 à liquidity_ratio_1 + classe LogisticRegression
- Fichier2.3parametres : comparaison capital_ratio_2 à deposits_risks_1
- Fichier2.4parametres : comparaison deposits_risks_1 à securities
- Fichier2.5parametres : comparaison capital_ratio_2 à securities
- Fichier2.6parametres : : comparaison capital_ratio_3 à deposits_risks_1 et securities
- Fichier2.7parametres : comparaison capital_ratio_2 à liquidity_ratio_2 et deposits_risks_1
- Fichier3_logisticRegression : fonction LogisticRegression

Introduction

Le manuel d'utilisateur suivant détaille et explique le code utilisé et remis pour répondre au sujet suivant, Forecast of banks 'default and firms returns using their respective historical financial statements.

L'objectif à terme du sujet est de prédire les comportements à risque des banques françaises entre 1900 et 1940.

L'objectif du code est livré est dans un premier temps d'extraire à des fins d'exploitation les données du fichier csv 'Data_table 3', remis par M.Riva.

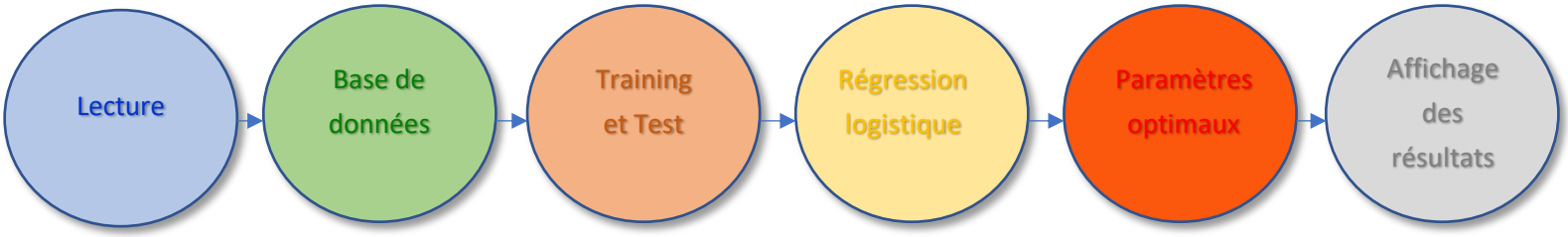
Il s'agit ensuite d'implémenter les équations économiques suivantes, :

- Capital Ratio
- Securities
- Liquidité
- Risque de dépôts
- Risque de crédits

Une fois les équations stockées en mémoire, il faut construire le modèle algorithmique de prédiction à l'aide d'une régression logistique.

Le modèle doit nous permettre d'obtenir des résultats similaires à ceux de Allen N Berger et Christa HS Bouwman dans leurs travaux "How does capital affect bank performance during financial crisis ? "

Les derniers résultats que l'on obtient doivent nous permettre d'expliquer plus en détails le rôle joué par catégories économiques ci-dessus



Lecture

Comprendre les caractéristiques des données permet de construire un modèle pertinent car d'une précision supérieure.

A / Code

```
21  #-------------------------------------------------------------------
22                              #LECTURE DU FICHIER EXCEL
23
24  print("PARTIE 1 : Resprésentation graphique des données")
25  #Lecture du fichier excel
26  document = xlrd.open_workbook("Data_table_3.xlsx")
27  fichier = "Data_table_3.xlsx"
28  sheet = "Sheet1"
29  df = pd.read_excel(io=fichier, sheet_name=sheet)
```

- Ligne 26 : ***accès fichier*** *à l'aide de la fonction open_workbook de la bibliothèque xlrd*
- Ligne 29 : ***ouverture de la feuille excel contenant les données*** *à l'aide de la fonction read_excel de la bibliothèque pandas*

B / Fonctions et modules utilisés

- Xlrd : bibliothèque de lecture de données et d'informations de mise en forme à partir de fichiers Excel au format xls.
  - Open_workbook : ouvre un fichier de feuille de calcul pour l'extraction des données
    - Paramètres : filename =chemin du fichier devant être ouvert

- Pandas : bibliothèque open source fournissant des structures de données et des outils d'analyse pour Python.
  - pd.read_excel : lit un fichire excel dans un dataframe
    - io : fichier à lire
    - sheet_name : feuille contenant les données à lire

Création d'une base de données

A / Code

```
35  ligne_debut = 1
36  ligne_fin = 6000
37  ligne = ligne_fin - ligne_debut
38
39  df_new = df[['id', 'referenceyear', 'tot_assets', 'capital', 'capital_surplus',
40               'paidincapital', 'tot_deposits', 'cash', 'comm_portfolio',
41               'securities', 'tot_credit']]
42
43  #df_data = df_new.head(ligne)
44  df_data = df_new[ligne_debut:ligne_fin]
45
46  print('-------------------------------------')
47  print("Données à utiliser : \n{}".format(df_data))
48
49  df_new = df[['f1']]
50
51  #df_resultat = df_new.head(ligne)
52  df_resultat = df_new[ligne_debut:ligne_fin]
53
54  print('-------------------------------------')
55  print("Faillit ou non des banques : \n{}".format(df_resultat))
56
57
```

- Ligne 35 à 37 : *variables indiquant le* ***nombre de lignes du fichier à parcourir***
- Ligne 39 : ***extraction des données des colonnes*** *choisies. df est une variable de type dataframe contenant les données du fichier excel.*
  Par souci d'optimisation on extrait seulement les colonnes à utiliser
- Ligne 44 : ***on parcourt*** *des données de la* ***ligne 1 à 6000*** *des colonnes extraites dans df_data*
- Ligne 49 et 52 : *Raisonnement similaire aux ligne 39 et 44. Ici df_resultat comprendra les* ***résultats à prédire****, soit le risque de faillite égal à 0 ou 1.*

B / Propriétés utilisées

❖ Variable_de_type_dataframe[**de_index_a :a_index_b**] =>Accès aux valeurs des index 'de_index_a' à 'a_index_b'

➢ Exemple :

```
>>> df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
...                    columns=['a', 'b', 'c'])
>>> df2
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

Training et Test

A / Code

```
58  #-----------------------------------------------------------------------
59                              #CREATION DES DONNEES DE TRAINING ET DE TEST
60
61  #Choix du nombre de lignes qui vont etre sélectionnée -------------------------
62  calcul = 0.75*ligne
63  resultat = int(round(calcul))
64  #resultat = 75
65  fin = ligne_fin
66
67  #Initialisation des paramètres -------------------------------------------------
68  capital_ratio_1 = (df_data['capital'])/df_data['tot_assets']
69  df_data['capital_ratio'] = capital_ratio_1
70
71  liquid_ratio_1 = df_data['cash']/df_data['tot_assets']
72  df_data['liquidity_ratio'] = liquid_ratio_1
73
74  #Replace the nan values by 0
75  df_data.fillna(value=0, inplace = True)
76  df_resultat.fillna(value=0, inplace = True)
77
78
79
80  #Training data -----------------------------------------------------------------
81  Training =  np.array([df_data['referenceyear'][0:resultat],
82                        df_data['capital_ratio'][0:resultat],
83                        df_data['liquidity_ratio'][0:resultat]])
84
85  #plt.plot(Training[0], Training[1], label='Total assets')
86  plt.scatter(Training[0], Training[1], label='Capital Ratio')
87  plt.scatter(Training[0], Training[2], label='liquidity ratio')
88  #plt.plot(Training[0], Training[7], label='real_capital_ratio')
89  #plt.plot(Training[0], Training[8], label='balence commerciale')
90  plt.legend()
91
92  #Test data ---------------------------------------------------------------------
93  Test = np.array([df_data['referenceyear'][resultat:fin],
94                        df_data['capital_ratio'][resultat:fin],
95                        df_data['liquidity_ratio'][resultat:fin]])
96
```

- Ligne 62 à 65 : *calcul du **nombre de lignes** représentant **75%** de la totalité*
- Ligne 68 à 72 : ***implémentation** des équations choisies.*
    - *Ex : à la ligne 98 capital ratio correspond à la division respectives des valeurs des index correspondants de la colonne capital par ceux de la colonne tot_assets. La valeur de la division est stockée dans capital_ratio_1, qui est ensuite ajoutée df_data.*
- Ligne 75 et 76 : ***traitement des données**.*
  On vérifie qu'il n'a pas de valeurs négatives ou nulles (NAN)

- Ligne 81 à 84 : *construction des **données de training**.*
  On créé un array contenant les valeurs des équations que l'on va utiliser et comparer. Elles correspondent à 75% des données d'input
- Ligne 85 à 80 : *affichage de ces données pour vérifier qu'elles ont les bonnes dimensions*
- Ligne 93 à 95 : *raisonnement identique qu'aux lignes 81 à 84, sauf que l'on prend les 25% dernières lignes comme **valeurs à tester**.*

B / Fonctions et modules utilisés

❖ Dataframe.fillna : remplace les valeurs nulles par une valeur imposée, ici 0.
  ➢ Value : valeur que vont prendre les NAN
  ➢ Inplace : permet de changer à la main les NAN grâce au paramètre value

❖ Numpy : bibliothèque qui fournit un objet de tableau multidimensionnel, divers objets dérivés (tels que des tableaux et des matrices masqués) et un assortiment de routines pour des opérations sur les tableaux.
  ➢ Np.array : créer un tableau à partir d'objets indiqués en paramètres, ici df_data ou df_resultat

❖ Matplotlib.pyplot : module de la bibliothèque matplotlib, inspirée de matlab, permettant l'affichage de données sous la forme de graphiques
  ➢ Plt.scatter : Un nuage de points de y par rapport à x avec une taille et / ou une couleur de marqueur variables, avec x et y les paramètres à afficher et color la couleur en string que l'on veut avoir
  ➢ Plt.show() : commande pour afficher le graphique dans le terminal

Construction du modele : Normalisation/Standardisation

La normalisation est une méthode de prétraitement des données qui permet de réduire la complexité des modèles.
D'un point de vue mathématique, la normalisation standardise la moyenne et l'écart type d'une distribution de données, ce qui permet de simplifier le problème (l'hypothèse).

$$Xnormalisé = \frac{X - m}{\sigma}$$

Avec m : moyenne et σ : écart-type

A / Code

```
97    #Training matrice ----------------------------------------------------------
98    Xtrain = Training
99    Xtrain = Xtrain.T
100   Ytrain = np.array([df_resultat['f1'][0:resultat]])
101   Ytrain = Ytrain.T
102
103   #Test matrice --------------------------------------------------------------
104   Xtest = Test
105   Xtest = Xtest.T
106   Ytest = np.array([df_resultat['f1'][resultat:fin]])
107   Ytest = Ytest.T
108
109   #Ajout de la colonne de 1 à la matrice Xtrain -------------------------------
110   p = np.ones([len(Xtrain),1])
111   Xtrain = np.append(p, Xtrain, axis = 1)
112
113   #Ajout de la colonne de 1 à la matrice Xtest---------------------------------
114   p = np.ones([len(Xtest),1])
115   Xtest = np.append(p, Xtest, axis = 1)
116
117
```

- Ligne 98 à 101 : *création des **matrices de training**. On applique la transposée de Xtrain et Ytrain pour que les dimensions puissent permettre par la suite une multiplication vectorielle*
- Ligne 104 à 107 : *création des **matrices tests** dans le même raisonnement*
- Ligne 110 à 115 : *ajout d'une c**olonne de 1** pour appliquer la fonction de coût de type cfs par la suite (voir partie suivante)*

B / Fonctions et modules utilisés

- Numpy
  - Array.T : fonction transposée de la matrice
  - Np.ones : colonne de 1
    - Nombre de lignes de la matrice
    - Nombre de colonnes de la matrice
  - Np.append : ajout d'éléments à la matrice/tableau
    - Ce qu'on ajoute
    - Dans quelle matrice
    - Si on insère en colonne ou ligne (si 1 alors colonne)

Construction du modele : Régression/Logistique

La régression logistique est un algorithme de classification d'apprentissage supervisé utilisé pour prédire la probabilité d'un événement. Il n'y a que 2 classes possibles

Un modèle de régression logistique prédit la probabilité de Y = 1 en fonction de X. C'est l'un des algorithmes ML les plus simples qui peut être utilisé pour divers problèmes de classification.

On doit définir l'hypothèse, la fonction de coût et l'erreur. L'hypothèse est déterminée à partir de la fonction sigmoid.

$$S(x) = \frac{1}{1 + e^{-x}}$$

Ces 3 fonctions permettent d'obtenir la prédiction des résultats et leur précision.

```
118  #--------------------------------------------------------------------------------
119  #---------------------------------DEFINITION FONCTIONS CFS-----------------------
120
121  #Fonction hypothèse
122  def Hypothese(X, Theta):
123      return X.dot(Theta)
124
125  #Fonction CFS
126  def CFS(X, Y):
127      p1 = np.dot(X.T,X)
128      p2 = np.dot(X.T, Y)
129      #Theta = np.linalg.pinv(p1).dot(p2)
130      Theta =  np.linalg.pinv(p1)
131      Theta = Theta.dot(p2)
132      #Theta = Theta[~np.isnan(Theta)]
133      return Theta
134
135  #Fonction Erreur
136  def Erreur(x,y,theta):
137      ypred = np.dot(x,theta)
138      I = len(y)
139      # boucle for pour parcourir les matrices
140      for i in range (I):
141          erreur = (1/I) * np.sum(np.square(ypred-y))  #MSE (mean square error)
142      return erreur
143
```

- Ligne 122 : *fonction de **l'hypothèse***
  Nous prenons ici l'hypothèse = multiplication vectorielle de X par theta où theta est le poids du modèle.

- Ligne 126 *: fonction de coût de la forme **CFS**. Description ligne par ligne*
  - *1/Produit matricielle de la transposée de X par X*
  - *2/Produit matriciel de la transposée de X par Y*
  - *3/Theta prend la valeur de l'inverse de 1/*
  - *4/Theta prend la valeur du produit matriciel de 3/ par 2/*

- Ligne 136 : *fonction retournant la probabilité de **l'erreur**. Description de la fonction*
  - *1/Instanciation de la valeur prédite du résultat*
  - *2/Parcours toutes les données de Ytrain*
  - *3/ Formule du MSE (voir définition plus haut).*

Construction du modele : Paramètres optimaux

```
143  #------------------------------PARAMETRES OPTIMAUX-----------------------------
144
145  #ori_data.dropna(inplace=True)
146  Theta_F = CFS(Xtrain, Ytrain)
147  #Theta_F = Theta_F[~np.isnan(Theta_F)]
148  #x = x[~numpy.isnan(x)]
149
150
151  print(Theta_F)
152
153  #Prédiction des données de training-------------------------------------------
154  Yprediction = Hypothese(Xtrain, Theta_F)
155  error = Erreur(Xtrain, Ytrain, Theta_F)
156  print("Erreur : ", error)
157
158  #Affichage des données--------------------------------------------------------
159  fig1 = plt.figure()
160  ax1 = fig1.gca(projection='3d')
161  ax1.scatter3D(Xtrain[:,1], Xtrain[:,2], Ytrain, color= 'green')
162  ax1.scatter3D(Xtrain[:,1], Xtrain[:,2], Yprediction, color= 'red')
163  ax1.set_xlabel('x1', fontweight = 'bold')
164  ax1.set_ylabel('x2', fontweight = 'bold')
165  ax1.set_zlabel('y', fontweight = 'bold')
166  plt.show()
167
```

A / Code

- Ligne 146 : ***Création de Theta***
- Ligne 154 : *Résultats prédits, soit la probabilité de faillite, grâce à l'hypothèse*
- Ligne 155 et 156 : *Grâce à la variable error, estimation de la précision du modèle.*
- Ligne 159 à 166 : *Affichage des résultats prédits comparés à leur probabilité de faillite, eux-mêmes comparés à la troisième colonne de Xtrain. La 1ere étant la date et la 2eme la première équation.*

Annexe : classe Regression Logistique

Nous précisons que l'annexe sera moins détaillée que le code jusqu'alors présenté. En effet il s'agit de code pour l'instant non fonctionnel, ayant pour but de pouvoir être réutilisé lors de la suite du projet.

```python
class LogisticRegression:
    def __init__(self, lr = 0.01, num_iter = 100000, fit_intercept = True, verbose = False):
        self.lr = lr
        self.num_iter = num_iter
        self.fit_intercept = fit_intercept
        self.verbose = verbose

    def __add_intercept(self, X):
        intercept = np.ones((X.shape[0], 1))
        return np.concatenate((intercept, X), axis=1)

    def __sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def __loss(self, h, y):
        return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()

    def fit(self, X, y):
        if self.fit_intercept:
            X = self.__add_intercept(X)
            self.theta = np.zeros(X.shape[1])
            for i in range(self.num_iter):
                z = np.dot(X, self.theta)
                h = self.__sigmoid(z)
                gradient = np.dot(X.T, (h - y)) / y.size
                self.theta -= self.lr * gradient

                z = np.dot(X, self.theta)
                h = self.__sigmoid(z)
                loss = self.__loss(h, y)

                if(self.verbose ==True and i % 10000 == 0):
                    print(f'loss: {loss} \t')

    def predict_prob(self, X):
        if self.fit_intercept:
            X = self.__add_intercept(X)
            return self.__sigmoid(np.dot(X, self.theta))

    def predict(self, X):
        return self.predict_prob(X).round()
```

La classe LogisticRegression regroupe les différentes étapes de la régression logistique soit forme de fonctions.

D'abord on définit les attributs dans le constructeur. Il s'agit d'alpha de la 'logistic regression', du nombre d'itérations de l'algorithme (nombre de fois que le scénario tournera afin d'obtenir une valeur plus précise de theta).

La fonction __add_intercept permet d'ajouter la colonne pour modifier les dimensions de Xtrain afin de pouvoir effectuer le calcul de coût.

La fonction fit permet de normaliser Xtrain et Ytrain, puis Xtest et Ytest.
Elle permet aussi de trouver la valeur de theta à l'aide de la descente de gradient.

La fonction predict_prob renvoie les résultats prédits et predict arrondi predict_prob à l'unité.

```
#Variables ------------------------------------------------------------------
model = LogisticRegression(lr = 0.1, num_iter = 300000)
model = model.fit(Xtrain, Ytrain)
preds = model.predict(Xtrain)
```

```
  File "D:\ECE\Mineur Recherche\fichier2.2_parametres.py", line 262, in <module>
    model = model.fit(Xtrain, Ytrain)

  File "D:\ECE\Mineur Recherche\fichier2.2_parametres.py", line 242, in fit
    self.theta -= self.lr * gradient

ValueError: operands could not be broadcast together with shapes (5,) (5,4499) (5,)
```

L'appel à la classe LogisticRegression renvoie une erreur de dimension que nous n'avons pas pu résoudre

Annexe : fonction Regression Logistique

```python
def Sigmoid(z):
    result = float(1.0 / float((1.0 + math.exp(-1.0*z))))
    return result

#Hypothèse ----------------------------------------------------------------
def Hypothese(theta, x):
    z = 0
    for i in range(len(theta)):
        z += x[i]*theta[i]
    return Sigmoid(z)

#Fonction de Coût ----------------------------------------------------------
def costFunction(X,Y,theta,m):
    sumOfErrors = 0
    for i in range(m):
        xi = X[i]
        hi = Hypothese(theta,xi)
        if Y[i] == 1:
            error = Y[i] * math.log(hi)
        elif Y[i] == 0:
            error = (1-Y[i]) * math.log(1-hi)
        sumOfErrors += error
    const = -1/m
    J = const * sumOfErrors
    print ('cost is ', J )
    return J

#Composé gradient pour chaque valeur de Théta ----------------------------
def costFunctionDerivative(X,Y,theta,j,m,alpha):
    sumErrors = 0
    for i in range(m):
        xi = X[i]
        xij = xi[j]
        hi = Hypothese(theta,X[i])
        error = (hi - Y[i])*xij
        sumErrors += error
    m = len(Y)
    constant = float(alpha)/float(m)
    J = constant * sumErrors
    return J

#Gradient descent ----------------------------------------------------------
def gradientDescent(X,Y,theta,m,alpha):
    new_theta = []
    constant = alpha/m
    for j in range(len(theta)):
        CFDerivative = costFunctionDerivative(X,Y,theta,j,m,alpha)
        new_theta_value = theta[j] - CFDerivative
        new_theta.append(new_theta_value)
    return new_theta

```

```
185 #Fonction logistique ----------------------------------------------
186 def Logistic_Regression(X,Y,alpha,theta,num_iters):
187     m = len(Y)
188     for x in range(num_iters):
189         new_theta = gradientDescent(X,Y,theta,m,alpha)
190         theta = new_theta
191         if x % 100 == 0:
192             costFunction(X,Y,theta,m)
193             print ('theta ', theta)
194             print ('cost is ', costFunction(X,Y,theta,m))
195
```

Le code ci-dessus permet d'utiliser directement la régression logistique depuis la fonctionLogistic_Regression.
Les fonction de gradientDescent, Erreur, Sigmoid et Hypothese restent similaires à ce qui a été vu.

L'intérêt est de n'avoir qu'à appeler la fonction.
Toutefois elle également inutilisable pour l'instant suite à un problème de dimensions