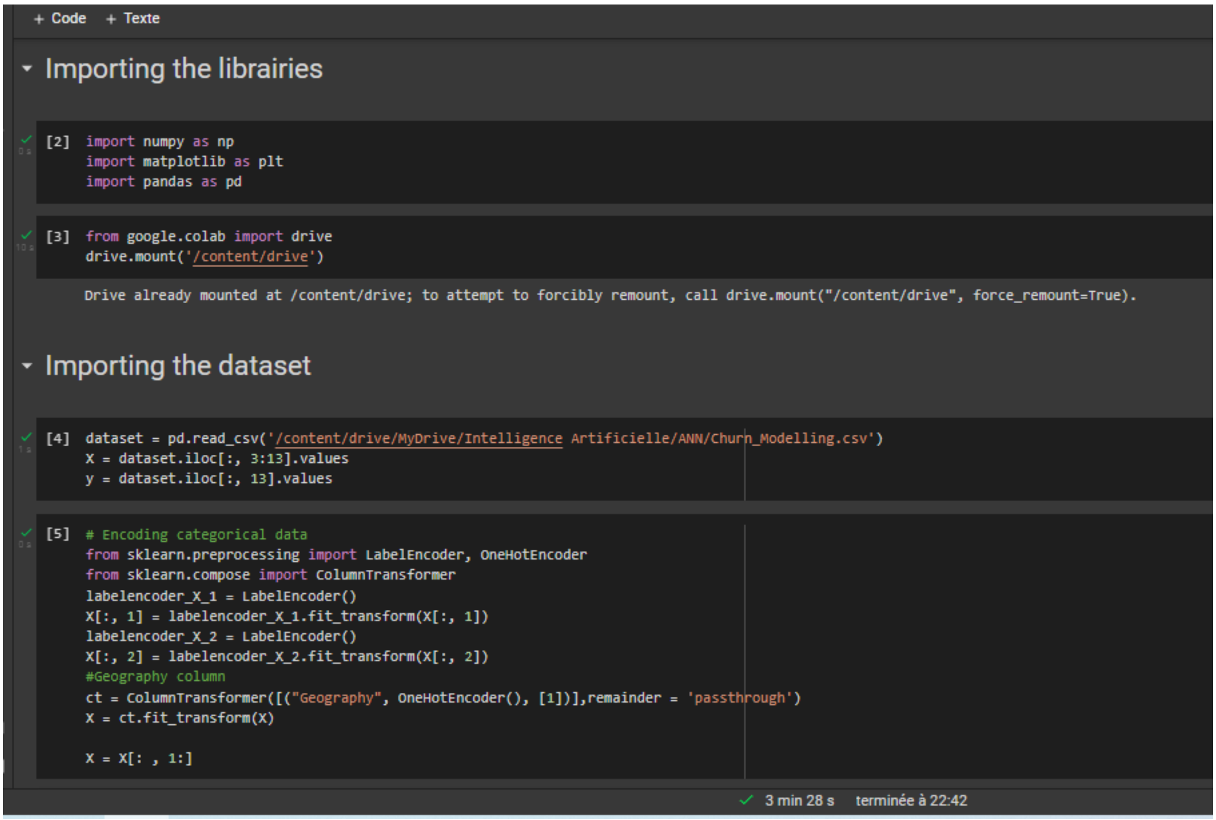
Intelligence Artificielle

COMPTE RENDU DES TPS

1. Les réseaux de neurones artificiels (ANN) :

a. Exécuter le code (captures écrans)



+ Code + Texte

Importing the librairies

```
[2] import numpy as np
    import matplotlib as plt
    import pandas as pd
```

```
[3] from google.colab import drive
    drive.mount('/content/drive')

    Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Importing the dataset

```
[4] dataset = pd.read_csv('/content/drive/MyDrive/Intelligence Artificielle/ANN/Churn_Modelling.csv')
    X = dataset.iloc[:, 3:13].values
    y = dataset.iloc[:, 13].values
```

```
[5] # Encoding categorical data
    from sklearn.preprocessing import LabelEncoder, OneHotEncoder
    from sklearn.compose import ColumnTransformer
    labelencoder_X_1 = LabelEncoder()
    X[:, 1] = labelencoder_X_1.fit_transform(X[:, 1])
    labelencoder_X_2 = LabelEncoder()
    X[:, 2] = labelencoder_X_2.fit_transform(X[:, 2])
    #Geography column
    ct = ColumnTransformer([("Geography", OneHotEncoder(), [1])],remainder = 'passthrough')
    X = ct.fit_transform(X)

    X = X[: , 1:]
```

3 min 28 s terminée à 22:42

```python
# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

#2 construire le reseau de neurone ANN
#importation des modules keras
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout

#Etape 1 : Initiation du ANN
classifier = Sequential()

#Ajouter la couche d'entrée et une couche cachée
classifier.add(Dense(units=6, activation="relu", kernel_initializer="uniform",input_dim=11))

classifier.add(Dropout(rate=0.1))
#Ajouter une deuxieme couche cachée
classifier.add(Dense(units=6, activation="relu", kernel_initializer="uniform"))

classifier.add(Dropout(rate=0.1))

#Ajouter une couche de sortie
classifier.add(Dense(units=1, activation="sigmoid", kernel_initializer="uniform"))

#Compilation
classifier.compile(optimizer="adam", loss= "binary_crossentropy", metrics=["accuracy"])

#Entrainement du reseau
```

✓ 3 min 28 s terminée à 22:42

```python
#Entrainement du reseau
classifier.fit(X_train,y_train, batch_size=10, epochs=100)

#prediction the test
y_pred = classifier.predict(X_test)
y_pred = (y_pred > 0.5)

#♣Confusion matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test , y_pred)
```

```
Epoch 82/100
800/800 [============================] - 1s 2ms/step - loss: 0.3661 - accuracy: 0.8486
Epoch 83/100
800/800 [============================] - 1s 2ms/step - loss: 0.3657 - accuracy: 0.8471
Epoch 84/100
800/800 [============================] - 1s 2ms/step - loss: 0.3641 - accuracy: 0.8486
Epoch 85/100
800/800 [============================] - 1s 2ms/step - loss: 0.3657 - accuracy: 0.8460
Epoch 86/100
800/800 [============================] - 1s 2ms/step - loss: 0.3634 - accuracy: 0.8462
Epoch 87/100
800/800 [============================] - 1s 2ms/step - loss: 0.3650 - accuracy: 0.8456
Epoch 88/100
800/800 [============================] - 1s 2ms/step - loss: 0.3619 - accuracy: 0.8500
Epoch 89/100
800/800 [============================] - 1s 2ms/step - loss: 0.3666 - accuracy: 0.8445
Epoch 90/100
800/800 [============================] - 1s 2ms/step - loss: 0.3636 - accuracy: 0.8511
Epoch 91/100
800/800 [============================] - 1s 2ms/step - loss: 0.3626 - accuracy: 0.8486
Epoch 92/100
800/800 [============================] - 1s 2ms/step - loss: 0.3629 - accuracy: 0.8447
Epoch 93/100
800/800 [============================] - 1s 2ms/step - loss: 0.3636 - accuracy: 0.8487
Epoch 94/100
800/800 [============================] - 1s 2ms/step - loss: 0.3649 - accuracy: 0.8478
Epoch 95/100
800/800 [============================] - 1s 2ms/step - loss: 0.3660 - accuracy: 0.8462
Epoch 96/100
800/800 [============================] - 1s 2ms/step - loss: 0.3626 - accuracy: 0.8484
Epoch 97/100
800/800 [============================] - 1s 2ms/step - loss: 0.3654 - accuracy: 0.8475
Epoch 98/100
800/800 [============================] - 1s 2ms/step - loss: 0.3637 - accuracy: 0.8499
Epoch 99/100
800/800 [============================] - 1s 2ms/step - loss: 0.3600 - accuracy: 0.8512
Epoch 100/100
800/800 [============================] - 1s 2ms/step - loss: 0.3609 - accuracy: 0.8486
```

✓ 3 min 28 s terminée à 22:42

b. La performance de votre model (accuracy) à l'aide la matrice de confusion est 0.8486

c. Prévoir si le client ci-dessus va quitter ou rester dans la banque :

Pays : France Score de crédit : 412.

Genre : Masculin Âge : 45 ans

Durée depuis entrée dans la banque : 2 ans

Balance : 280000 € Nombre de produits : 3

Carte de crédit ? Non

Membre actif ? : Oui

Salaire estimé : 60000 €

```
[17] client_test = np.array([[0.0 ,0.0 , 412 , 0 ,45 ,2 ,280000 , 3, 0 , 1, 60000]])
     my_predicition = classifier.predict(sc.transform(client_test))
     print(my_predicition)
     my_prediction = (my_predicition > 0.5)
     print(my_prediction)

     [[0.99997544]]
     [[ True]]
```

d. Proposer une amélioration de votre model à l'aide de GridsearchCV

- Capturer les hyper paramètres que vous voulez évaluer

```
#Evaluation de notre Modele

from  keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import StandardScaler

def build_classifier() :

    classifier = Sequential()

    classifier.add(Dense(units=6, activation="relu", kernel_initializer="uniform",input_dim=11))

    classifier.add(Dense(units=6, activation="relu", kernel_initializer="uniform"))

    classifier.add(Dense(units=1, activation="sigmoid", kernel_initializer="uniform"))

    classifier.compile(optimizer="adam", loss= "binary_crossentropy", metrics=["accuracy"])

    return classifier

classifier = KerasClassifier(build_fn=build_classifier, batch_size=10, epochs=100)

precision = cross_val_score(estimator= classifier, X=X, y=y , cv= 11 )

moyenne = precision.mean()
ecart_type = precision.std()

print(precision)
print(moyenne)
print(ecart_type)
```

- Entrainer votre model (Capture écran)

```python
#partie amelioration ANN

from  keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV

def build_classifier(optimizer) :

    classifier = Sequential()

    #Ajouter la couche d'entrée et une couche cachée
    classifier.add(Dense(units=6, activation="relu", kernel_initializer="uniform",input_dim=11))
    #Ajouter une deuxieme couche cachée
    classifier.add(Dense(units=6, activation="relu", kernel_initializer="uniform"))
    #Ajouter une couche de sortie
    classifier.add(Dense(units=1, activation="sigmoid", kernel_initializer="uniform"))
   #Compilation
    classifier.compile(optimizer= optimizer, loss= "binary_crossentropy", metrics=["accuracy"])

    return classifier

classifier = KerasClassifier(build_fn=build_classifier)
parameters = {"batch_size" : [ 25, 32],
              "epochs" : [100 , 500] ,
              "optimizer" : ["adam","rmsprop"]}

gridsearch =GridSearchCV(estimator = classifier , param_grid= parameters , scoring="accuracy",cv=10)

gridsearch = gridsearch.fit(X_train, y_train)

best_parametrs = gridsearch.best_parametrs
best_precision = gridsearch.best_precision
```

- Quel est la durée de l'entrainement est : 6h24min59s
- Quel est la combinaison optimale que vous avez trouvée, best_parameters (Capture écran)
- Quel la performance et la précision de cette combinaison, best_precisions (Capture écran)

```
250/250 [==============================] - 1s 2ms/step - loss: 0.3983 - accuracy: 0.8370
Epoch 493/500
250/250 [==============================] - 1s 3ms/step - loss: 0.3983 - accuracy: 0.8356
Epoch 494/500
250/250 [==============================] - 1s 3ms/step - loss: 0.3987 - accuracy: 0.8359
Epoch 495/500
250/250 [==============================] - 1s 3ms/step - loss: 0.3989 - accuracy: 0.8356
Epoch 496/500
250/250 [==============================] - 1s 3ms/step - loss: 0.3989 - accuracy: 0.8364
Epoch 497/500
250/250 [==============================] - 1s 3ms/step - loss: 0.3988 - accuracy: 0.8353
Epoch 498/500
250/250 [==============================] - 1s 3ms/step - loss: 0.3986 - accuracy: 0.8354
Epoch 499/500
250/250 [==============================] - 1s 3ms/step - loss: 0.3988 - accuracy: 0.8361
Epoch 500/500
250/250 [==============================] - 1s 3ms/step - loss: 0.3990 - accuracy: 0.8360
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-414bd40e1b13> in <module>()
     59 gridsearch = gridsearch.fit(X_train, y_train)
     60 
---> 61 best_parametrs = gridsearch.best_parametrs
     62 best_precision = gridsearch.best_precision

AttributeError: 'GridSearchCV' object has no attribute 'best_parametrs'

SEARCH STACK OVERFLOW
```

3. Les réseaux de neurones à convolution (CNN)

e. Exécuter le code (captures écrans)

```python
   # Importing libraries
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense
   # Initialising the CNN
classifier = Sequential()
   # Step 1 - Convolution
classifier.add(Conv2D(32, (3, 3), input_shape = (64, 64, 3), activation = 'relu'))
   # Step 2 - Pooling
classifier.add(MaxPooling2D(pool_size = (2, 2)))
# Adding a second convolutional layer
#classifier.add(Conv2D(32, (3, 3), activation = 'relu'))
#classifier.add(MaxPooling2D(pool_size = (2, 2)))
   # Step 3 - Flattening
classifier.add(Flatten())
   # Step 4 - Full connection
classifier.add(Dense(units = 128, activation = 'relu'))
classifier.add(Dense(units = 1, activation = 'sigmoid'))
# Compiling the CNN
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy',metrics = ['accuracy'])
```

```python
   # Part 2 - Fitting the CNN to the images
from keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)
test_datagen = ImageDataGenerator(rescale = 1./255)

training_set = train_datagen.flow_from_directory('/content/drive/MyDrive/Intelligence Artificielle/CNN/dataset/training_set'    ,
                                                 target_size = (64, 64),
                                                 batch_size = 32,
                                                 class_mode = 'binary' )

test_set = test_datagen.flow_from_directory('/content/drive/MyDrive/Intelligence Artificielle/CNN/dataset/test_set',
                                            target_size = (64, 64),
                                            batch_size = 32,
                                            class_mode = 'binary')
classifier.fit(training_set,
                         steps_per_epoch = 250,
                         epochs = 25,
                         validation_data = test_set,
                         validation_steps = 63)
```

```
Found 8000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
Epoch 1/25
250/250 [==============================] - 1581s 6s/step - loss: 0.7085 - accuracy: 0.5819 - val_loss: 0.6803 - val_accuracy: 0.5630
Epoch 2/25
250/250 [==============================] - 82s 330ms/step - loss: 0.6203 - accuracy: 0.6610 - val_loss: 0.5953 - val_accuracy: 0.6820
```

```
Epoch 3/25
250/250 [==============================] - 82s 326ms/step - loss: 0.5995 - accuracy: 0.6799 - val_loss: 0.5935 - val_accuracy: 0.6775
Epoch 4/25
250/250 [==============================] - 81s 325ms/step - loss: 0.5744 - accuracy: 0.7003 - val_loss: 0.5603 - val_accuracy: 0.7080
Epoch 5/25
250/250 [==============================] - 83s 330ms/step - loss: 0.5572 - accuracy: 0.7156 - val_loss: 0.5532 - val_accuracy: 0.7145
Epoch 6/25
250/250 [==============================] - 81s 325ms/step - loss: 0.5458 - accuracy: 0.7201 - val_loss: 0.5742 - val_accuracy: 0.7005
Epoch 7/25
250/250 [==============================] - 81s 326ms/step - loss: 0.5338 - accuracy: 0.7312 - val_loss: 0.5463 - val_accuracy: 0.7265
Epoch 8/25
250/250 [==============================] - 81s 323ms/step - loss: 0.5259 - accuracy: 0.7401 - val_loss: 0.5337 - val_accuracy: 0.7350
Epoch 9/25
250/250 [==============================] - 82s 330ms/step - loss: 0.5140 - accuracy: 0.7445 - val_loss: 0.5400 - val_accuracy: 0.7385
Epoch 10/25
250/250 [==============================] - 81s 326ms/step - loss: 0.5088 - accuracy: 0.7476 - val_loss: 0.5222 - val_accuracy: 0.7420
Epoch 11/25
250/250 [==============================] - 81s 323ms/step - loss: 0.4974 - accuracy: 0.7554 - val_loss: 0.5128 - val_accuracy: 0.7540
Epoch 12/25
250/250 [==============================] - 84s 336ms/step - loss: 0.4917 - accuracy: 0.7575 - val_loss: 0.5217 - val_accuracy: 0.7435
Epoch 13/25
250/250 [==============================] - 82s 326ms/step - loss: 0.4873 - accuracy: 0.7624 - val_loss: 0.5993 - val_accuracy: 0.7065
Epoch 14/25
250/250 [==============================] - 82s 328ms/step - loss: 0.4853 - accuracy: 0.7636 - val_loss: 0.6092 - val_accuracy: 0.7015
Epoch 15/25
250/250 [==============================] - 83s 332ms/step - loss: 0.4762 - accuracy: 0.7736 - val_loss: 0.5253 - val_accuracy: 0.7555
Epoch 16/25
250/250 [==============================] - 81s 324ms/step - loss: 0.4676 - accuracy: 0.7714 - val_loss: 0.5783 - val_accuracy: 0.7205
Epoch 17/25
250/250 [==============================] - 81s 323ms/step - loss: 0.4708 - accuracy: 0.7721 - val_loss: 0.5019 - val_accuracy: 0.7655
```

```
Epoch 18/25
250/250 [==============================] - 82s 327ms/step - loss: 0.4640 - accuracy: 0.7697 - val_loss: 0.5091 - val_accuracy: 0.7586
Epoch 19/25
250/250 [==============================] - 82s 327ms/step - loss: 0.4607 - accuracy: 0.7788 - val_loss: 0.5910 - val_accuracy: 0.7345
Epoch 20/25
250/250 [==============================] - 81s 324ms/step - loss: 0.4575 - accuracy: 0.7815 - val_loss: 0.5741 - val_accuracy: 0.7185
Epoch 21/25
250/250 [==============================] - 82s 328ms/step - loss: 0.4512 - accuracy: 0.7870 - val_loss: 0.5062 - val_accuracy: 0.7645
Epoch 22/25
250/250 [==============================] - 82s 329ms/step - loss: 0.4547 - accuracy: 0.7837 - val_loss: 0.5299 - val_accuracy: 0.7590
Epoch 23/25
250/250 [==============================] - 81s 326ms/step - loss: 0.4454 - accuracy: 0.7855 - val_loss: 0.5347 - val_accuracy: 0.7565
Epoch 24/25
250/250 [==============================] - 82s 328ms/step - loss: 0.4431 - accuracy: 0.7851 - val_loss: 0.5199 - val_accuracy: 0.7605
Epoch 25/25
250/250 [==============================] - 82s 328ms/step - loss: 0.4403 - accuracy: 0.7880 - val_loss: 0.5248 - val_accuracy: 0.7615
<keras.callbacks.History at 0x7f9789e7c490>
```

f. La performance de votre model (accuracy) est 0.7615
g. Tester votre model sur 2 images de chat et 2 images de chien, capturer vos résultats

<u>Pour le 1er chat</u>

```python
import numpy as np
from keras.preprocessing import image
test_image=image.load_img('/content/drive/MyDrive/Intelligence Artificielle/CNN/dataset/exercice_prediction/chien_chat_6.jpg',target_size=(64,64)
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = classifier.predict(test_image)
print(training_set.class_indices)
if result[0][0] == 1:
    prediction = 'dog'
else:
    prediction = 'cat'
print(prediction)
```

```
{'cats': 0, 'dogs': 1}
cat
```

<u>Pour le 2e chat</u>

<u>Chat11</u>

```python
import numpy as np
from keras.preprocessing import image
test_image=image.load_img('chat11.jpg',target_size=(64,64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = classifier.predict(test_image)
print(training_set.class_indices)
if result[0][0] == 1:
    prediction = 'dog'
else:
    prediction = 'cat'
print(prediction)
```

```
{'cats': 0, 'dogs': 1}
cat
```

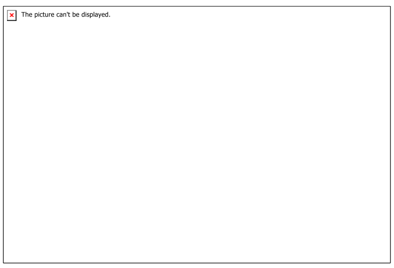
Pour le 1er chien

Chien11

```python
import numpy as np
from keras.preprocessing import image
test_image=image.load_img('chien11.jpg',target_size=(64,64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = classifier.predict(test_image)
print(training_set.class_indices)
if result[0][0] == 1:
    prediction = 'dog'
else:
    prediction = 'cat'
print(prediction)
```

```
{'cats': 0, 'dogs': 1}
dog
```

Pour le 2e chien



Chien12

```python
import numpy as np
from keras.preprocessing import image
test_image=image.load_img('chien12.webp',target_size=(64,64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = classifier.predict(test_image)
print(training_set.class_indices)
if result[0][0] == 1:
    prediction = 'dog'
else:
    prediction = 'cat'
print(prediction)
```

```
{'cats': 0, 'dogs': 1}
dog
```

1. Les réseaux de neurones récurrents (RNN)

h. Exécuter le code (captures écrans).

```python
import matplotlib.pyplot as plt
import pandas as pd

# Importing the training set
dataset_train = pd.read_csv('/content/drive/MyDrive/Intelligence Artificielle/RNN/Google_Stock_Price_Train.csv')
training_set = dataset_train.iloc[:, 1:2].values

# Virtualisation du resultat pour comprendre
q = pd.DataFrame(training_set)
print(q)


# Feature Scaling
from sklearn.preprocessing import MinMaxScaler
sc  = MinMaxScaler()
training_set_scaled = sc.fit_transform(training_set)

# Virtualisation du resultat pour comprendre
q = pd.DataFrame(training_set_scaled)
print(q)

# Creating a data structure with 60 timesteps and 1 output
X_train = []
y_train = []
for i in range (60, 1250):
  X_train.append(training_set_scaled[i-60:i, 0])
  y_train.append(training_set_scaled[i, 0])
X_train = np.array(X_train)
y_train = np.array(y_train)

# Visualisation  du resultat pour comprendre
q = pd.DataFrame(X_train)
```

```python
[1]
# Virtualisation du resultat pour comprendre
q = pd.DataFrame(training_set_scaled)
print(q)

# Creating a data structure with 60 timesteps and 1 output
X_train = []
y_train = []
for i in range (60, 1250):
  X_train.append(training_set_scaled[i-60:i, 0])
  y_train.append(training_set_scaled[i, 0])
X_train = np.array(X_train)
y_train = np.array(y_train)

# Visualisation  du resultat pour comprendre
q = pd.DataFrame(X_train)
print(q)
q = pd.DataFrame(y_train)
print(q)

# Reshaping
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
```

```
1256  0.937960
```

```
...        ...       ...       ...       ...       ...       ...       ...
1185  0.917665  0.911303  0.918111  0.931766  0.944025  0.936863  0.923395
1186  0.911303  0.918111  0.931766  0.944025  0.936863  0.923395  0.927766
1187  0.918111  0.931766  0.944025  0.936863  0.923395  0.927766  0.934445
1188  0.931766  0.944025  0.936863  0.923395  0.927766  0.934445  0.924939
1189  0.944025  0.936863  0.923395  0.927766  0.934445  0.924939  0.921069

            7         8         9  ...        50        51        52  \
0     0.065686  0.061091  0.066393  ...  0.052143  0.056124  0.058189
1     0.061091  0.066393  0.061426  ...  0.056124  0.058189  0.065407
2     0.066393  0.061426  0.074745  ...  0.058189  0.065407  0.068830
3     0.061426  0.074745  0.027978  ...  0.065407  0.068830  0.072438
4     0.074745  0.027978  0.023793  ...  0.068830  0.072438  0.079935
...        ...       ...       ...  ...       ...       ...       ...
1185  0.927766  0.934445  0.924939  ...  0.916009  0.913293  0.889798
1186  0.934445  0.924939  0.921069  ...  0.913293  0.889798  0.865894
1187  0.924939  0.921069  0.924381  ...  0.889798  0.865894  0.890301
1188  0.921069  0.924381  0.930482  ...  0.865894  0.890301  0.903360
1189  0.924381  0.930482  0.929905  ...  0.890301  0.903360  0.896421

            53        54        55        56        57        58        59
0     0.065407  0.068830  0.072438  0.079935  0.078466  0.080345  0.084977
1     0.068830  0.072438  0.079935  0.078466  0.080345  0.084977  0.086279
2     0.072438  0.079935  0.078466  0.080345  0.084977  0.086279  0.084716
3     0.079935  0.078466  0.080345  0.084977  0.086279  0.084716  0.074541
4     0.078466  0.080345  0.084977  0.086279  0.084716  0.074541  0.078838
...        ...       ...       ...       ...       ...       ...       ...
1185  0.865894  0.890301  0.903360  0.896421  0.917777  0.931766  0.941141
1186  0.890301  0.903360  0.896421  0.917777  0.931766  0.941141  0.957623
1187  0.903360  0.896421  0.917777  0.931766  0.941141  0.957623  0.964134
1188  0.896421  0.917777  0.931766  0.941141  0.957623  0.964134  0.964023
1189  0.917777  0.931766  0.941141  0.957623  0.964134  0.964023  0.969715

[1190 rows x 60 columns]
             0
0     0.086279
1     0.084716
2     0.074541
3     0.078838
4     0.072383
...        ...
1185  0.957623
1186  0.964134
1187  0.964023
1188  0.969715
1189  0.950778

[1190 rows x 1 columns]
```

i. Tester votre model sur le jeu de test (Google_Stock_Price_Test.csv), capturer le graphe

```
[2] # Part 2 - Building the RNN

    # Importing the keras libraries and the packages
    from keras.models import Sequential
    from keras.layers import Dense
    from keras.layers import LSTM
    from keras.layers import Dropout # pour casser la liaison entre les neurones afin d'eviter le surcharge

    # Initialising the RNN
    regressor  = Sequential()

    # Adding the first LSTM layer and some Dropout regularisation
    regressor.add(LSTM(units = 50, return_sequences = True, input_shape = (X_train.shape[1], 1)))
    regressor.add(Dropout(0.2))

    # Adding a second LSTM layer and some Dropout regularisation
    regressor.add(LSTM(units = 50, return_sequences = True))
    regressor.add(Dropout(0.2))

    # Adding a third LSTM layer and some Dropout regularisation
    regressor.add(LSTM(units = 50, return_sequences = True))
    regressor.add(Dropout(0.2))

    # Adding a fourth LSTM layer and some Dropout regularisation
    regressor.add(LSTM(units = 50))
    regressor.add(Dropout(0.2))
```

```
[2] # Adding the output layer
    regressor.add(Dense(units = 1))

    # compiling the RNN
    regressor.compile(optimizer = 'adam', loss = 'mean_squared_error')

    # fitting the RNN to the Training set
    regressor.fit(X_train, y_train, epochs = 100, batch_size = 32)

    38/38 [==============================] - 4s 111ms/step - loss: 0.0019
    Epoch 73/100
    38/38 [==============================] - 4s 111ms/step - loss: 0.0019
    Epoch 74/100
    38/38 [==============================] - 4s 111ms/step - loss: 0.0019
    Epoch 75/100
    38/38 [==============================] - 4s 112ms/step - loss: 0.0016
    Epoch 76/100
    38/38 [==============================] - 4s 110ms/step - loss: 0.0017
    Epoch 77/100
    38/38 [==============================] - 4s 109ms/step - loss: 0.0019
    Epoch 78/100
    38/38 [==============================] - 4s 114ms/step - loss: 0.0019
    Epoch 79/100
    38/38 [==============================] - 4s 118ms/step - loss: 0.0018
    Epoch 80/100
    38/38 [==============================] - 4s 116ms/step - loss: 0.0018
```

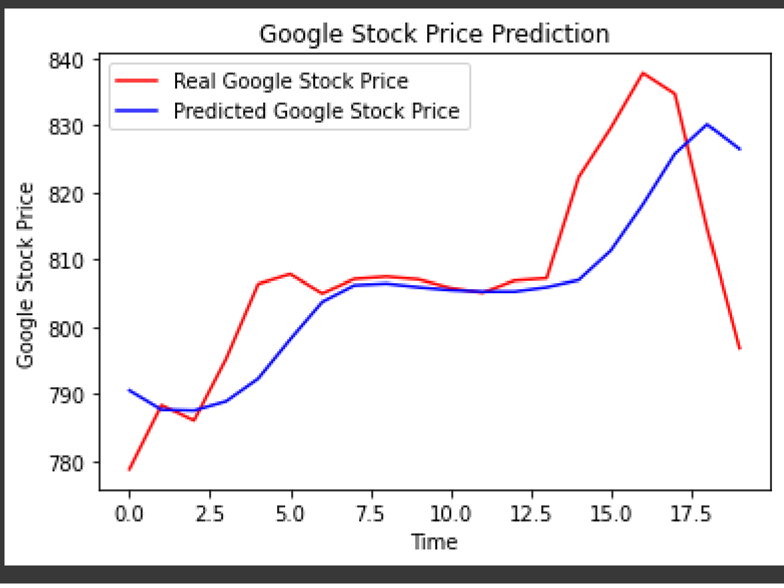
```
[2]  Epoch 81/100
38/38 [==============================] - 4s 112ms/step - loss: 0.0018
Epoch 82/100
38/38 [==============================] - 4s 115ms/step - loss: 0.0016
Epoch 83/100
38/38 [==============================] - 5s 127ms/step - loss: 0.0017
Epoch 84/100
38/38 [==============================] - 4s 115ms/step - loss: 0.0018
Epoch 85/100
38/38 [==============================] - 5s 127ms/step - loss: 0.0017
Epoch 86/100
38/38 [==============================] - 5s 135ms/step - loss: 0.0018
Epoch 87/100
38/38 [==============================] - 5s 134ms/step - loss: 0.0014
Epoch 88/100
38/38 [==============================] - 6s 152ms/step - loss: 0.0016
Epoch 89/100
38/38 [==============================] - 5s 142ms/step - loss: 0.0017
Epoch 90/100
38/38 [==============================] - 5s 130ms/step - loss: 0.0017
Epoch 91/100
38/38 [==============================] - 5s 137ms/step - loss: 0.0015
Epoch 92/100
38/38 [==============================] - 5s 138ms/step - loss: 0.0016
Epoch 93/100
38/38 [==============================] - 5s 125ms/step - loss: 0.0016
Epoch 94/100
38/38 [==============================] - 4s 114ms/step - loss: 0.0014
Epoch 95/100
38/38 [==============================] - 5s 129ms/step - loss: 0.0015
```

```
     Epoch 96/100
[2]  38/38 [==============================] - 5s 131ms/step - loss: 0.0015
Epoch 97/100
38/38 [==============================] - 4s 114ms/step - loss: 0.0014
Epoch 98/100
38/38 [==============================] - 4s 114ms/step - loss: 0.0017
Epoch 99/100
38/38 [==============================] - 4s 112ms/step - loss: 0.0016
Epoch 100/100
38/38 [==============================] - 4s 115ms/step - loss: 0.0014
<keras.callbacks.History at 0x7fde40ea4c90>
```

j. Prévoir la tendance du mois de janvier 2017, capturer le graphe

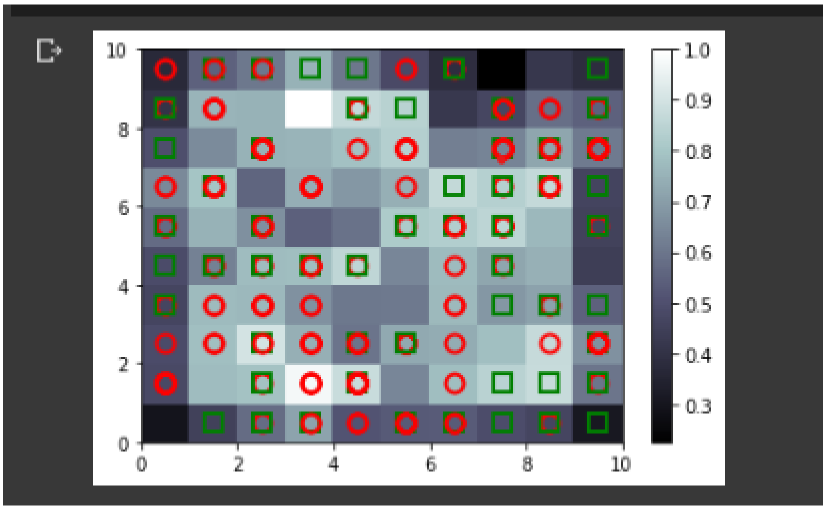
```python
# Part 3 - Making the predictions and visualising the results

#  Getting the real stock price of 2017
dataset_test = pd.read_csv('/content/drive/MyDrive/Intelligence Artificielle/RNN/Google_Stock_Price_Test.csv')
real_stock_price = dataset_test.iloc[:, 1:2].values

#  Getting the predicted stock price of 2017
dataset_total = pd.concat((dataset_train['Open'], dataset_test['Open']), axis = 0)
inputs  = dataset_total[len(dataset_total) - len(dataset_test) - 60:].values
inputs = inputs.reshape(-1, 1)
inputs = sc.transform(inputs)
X_test = []
for i in range (60, 80):
  X_test.append(inputs[i-60:i, 0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
predicted_stock_price = regressor.predict(X_test)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)

# Visualising the results
plt.plot(real_stock_price, color = 'red', label = 'Real Google Stock Price')
plt.plot(predicted_stock_price, color = 'blue', label = 'Predicted Google Stock Price')
plt.title('Google Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('Google Stock Price')
plt.legend()
plt.show()
```

2 s terminée à 23:57



2. Les cartes auto-adaptatifs (SOM)

k. Exécuter le code (capture écran de la carte).



l. Les IDs des clients qui sont susceptible d'être frauduleux sont : 15748432.0, 15696287.0, 15698749.0, 15815443.0, 15773776.0, 15757467.0