TP - Réseaux de Neurones

Mathieu SROUR - Omar ALLOUCH

Problème I

* $x_1 + x_2 + x_3 + x_4 = 0$

1ère itération : $P_1 \rightarrow 3 \quad \theta_1 = 1 \quad t_1 = 1$

$P_2 \rightarrow 1 \quad \theta_2 = 1 \quad t_2 = 0$

$e = -1$

$$W = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

$b = 0 - 1 = -1$

* $x_1 + x_3 + x_4 - 1 = 0$

2ème itération : $P_1 \rightarrow 1 \quad \theta_1 = 1 \quad t_1 = 1$

$P_2 \rightarrow -1 \quad \theta_2 = 0 \quad t_2 = 0$

$P_3 \rightarrow 2 \quad \theta_3 = 1 \quad t_3 = 1$

$P_4 \rightarrow 0 \quad \theta_4 = 1 \quad t_4 = 0$

$e = -1$

$$w = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

$b = -1 - 1 = -2$

* $x_1 + x_3 - 2 = 0$

3ème itération : $P_1 \rightarrow 0 \quad \theta_1 = 1 \quad t_1 = 1$

$P_2 \rightarrow -2 \quad \theta_2 = 0 \quad t_2 = 0$

$P_3 \rightarrow 0 \quad \theta_3 = 1 \quad t_3 = 1$

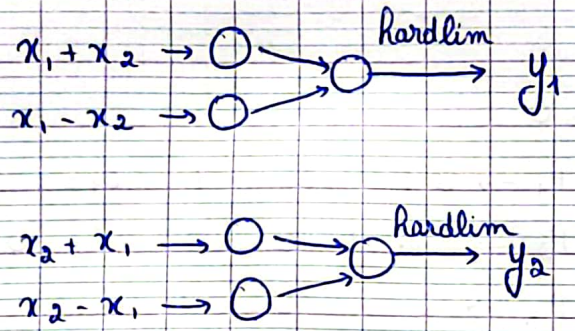
$P_4 \rightarrow -2 \quad \theta_4 = 0 \quad t_4 = 0$

$P_5 \rightarrow -2 \quad \theta_5 = 0 \quad t_5 = 0$

$$\boxed{W = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}} \qquad \boxed{b = -2}$$

Problème II

C'est un problème "XOR" entre les signes des variables d'entrées. Puisqu'on est limité à des réseaux simples à 2 entrées, il nous faut alors 2 réseaux, où la sortie de chacun est le signe d'une entrée.
On peut alors faire :



Puis, on agrége les sortie y_1 et y_2 par XOR :

$$y = y_1 \text{ XOR } y_2$$

Et on obtient la classification voulu.

En effet, le tableau de vérité de XOR est :

y_1	y_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Alors, si y_1 et y_2 étaient le signe de nos variables x_1 et x_2 on obtient la classification demandée.
Et nos 2 réseaux nous permet d'attaindre ça à l'aide de la fonction hardlim et les entrées comme spécifiés au-dessus. D'où le résultat.

Problem III

$y \sim B(p)$

$X|y=1 \sim N(\mu_1, \sigma^2)$

$X|y=0 \sim N(\mu_0, \sigma^2)$

1) $f(x) = P(y=1) f(X|y=1) + P(y=0) f(X|y=0)$

$$= p f(X|y=1) + (1-p) f(X|y=0)$$

$$= \frac{p}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu_1)^2}{2\sigma^2}} + \frac{(1-p)}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu_0)^2}{2\sigma^2}}$$

2) $p(x) = \mathbb{P}[y=1 / X=x] = \frac{\mathbb{P}[y=1; X=x]}{f(x)} = \frac{p f(X|y=1)}{f(x)}$

$$p(x) = \frac{\frac{p}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu_1)^2}{2\sigma^2}}}{\frac{p}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu_1)^2}{2\sigma^2}} + \frac{(1-p)}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu_0)^2}{2\sigma^2}}}$$

$$p(x) = \frac{p e^{-\frac{(x-\mu_1)^2}{2\sigma^2}}}{p e^{-\frac{(x-\mu_1)^2}{2\sigma^2}} + (1-p) e^{-\frac{(x-\mu_0)^2}{2\sigma^2}}}$$

3-) $\ln\left[\frac{p(x)}{1-p(x)}\right] = \ln[p(x)] - \ln[1-p(x)]$

$\longrightarrow \ln[p(x)] = \ln\left[\frac{p e^{-\frac{(x-\mu_1)^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}\, f(x)}\right] = \ln\left[p e^{-\frac{(x-\mu_1)^2}{2\sigma^2}}\right] - \ln\left[\sigma\sqrt{2\pi}\, f(x)\right]$

$= \ln(p) + \ln\left[e^{-\frac{(x-\mu_1)^2}{2\sigma^2}}\right] - \ln\left[\sigma\sqrt{2\pi}\, f(x)\right]$

$= \ln(p) - \frac{(x-\mu_1)^2}{2\sigma^2} - \ln\left[\sigma\sqrt{2\pi}\, f(x)\right]$

$\longrightarrow 1-p(x) = 1 - \frac{p e^{-\frac{(x-\mu_1)^2}{2\sigma^2}}}{p e^{-\frac{(x-\mu_1)^2}{2\sigma^2}} + (1-p) e^{-\frac{(x-\mu_0)^2}{2\sigma^2}}} = \frac{(1-p) e^{-\frac{(x-\mu_0)^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}\, f(x)}$

$$\rightarrow \ln[1-p(x)] = \ln(1-p) - \frac{(x-\mu_0)^2}{2\sigma^2} - \ln[\sigma\sqrt{2\pi}\, f(x)]$$

$$\rightarrow \ln\left[\frac{p(x)}{1-p(x)}\right] = \ln[p(x)] - \ln[1-p(x)]$$

$$= \ln(p) - \frac{(x-\mu_1)^2}{2\sigma^2} - \ln[\sigma\sqrt{2\pi}\, f(x)] - \left[\ln(1-p) - \frac{(x-\mu_0)^2}{2\sigma^2} - \ln[\sigma\sqrt{2\pi}\, f(x)]\right]$$

$$= \ln(p) - \ln(1-p) + \left[\frac{(x-\mu_0)^2}{2\sigma^2} - \frac{(x-\mu_1)^2}{2\sigma^2}\right]$$

$$= \ln\left(\frac{p}{1-p}\right) + \frac{1}{2\sigma^2}\left[x^2 + \mu_0^2 - 2x\mu_0 - (x^2 + \mu_1^2 - 2x\mu_1)\right]$$

$$= \ln\left(\frac{p}{1-p}\right) + \frac{1}{2\sigma^2}\left[\mu_0^2 - \mu_1^2 - 2x(\mu_0 - \mu_1)\right]$$

$$= \ln\left[\frac{p}{1-p}\right] + \frac{\mu_0^2 - \mu_1^2}{2\sigma^2} + x\frac{(\mu_1 - \mu_0)}{\sigma^2} = \beta_0 + \beta_1 x.$$

$$\beta_0 = \ln\left[\frac{p}{1-p}\right] + \frac{\mu_0^2 - \mu_1^2}{2\sigma^2} \qquad \beta_1 = \frac{\mu_1 - \mu_0}{\sigma^2}$$

$$\ln\left[\frac{p(x)}{1-p(x)}\right] = \beta_0 + \beta_1 x$$

$$\frac{p(x)}{1-p(x)} = e^{\beta_0 + \beta_1 x}$$

$$p(x) = (1-p(x))\, e^{\beta_0 + \beta_1 x}$$

$$p(x) + p(x)\, e^{\beta_0 + \beta_1 x} = e^{\beta_0 + \beta_1 x}$$

$$p(x) = \frac{e^{\beta_0 + \beta_1 x}}{1 + e^{\beta_0 + \beta_1 x}} = \frac{e^{\beta_0 + \beta_1 x}}{e^{\beta_0 + \beta_1 x}} \cdot \frac{1}{\frac{1}{e^{\beta_0 + \beta_1 x}} + 1} = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

4) En pratique, la détermination de $p(x)$ en utilisant l'expression de la partie précédente est difficile car $p(x)$ dépend de β_0 et β_1, qui, à leur tour dépendent de p, μ_0, μ_1 et σ qui sont inconnues ~~et sont~~

De plus, on assume une densité ~~d'une~~ gaussienne pour ~~$X|Y=1$ et $X|Y=0$~~ $X|Y=1$ et $X|Y=0$, ce qui n'est pas toujours le cas, spécialement si la taille de notre donnée n'est pas suffisamment grande.

5) On estime β_0 et β_1 par la méthode de maximum de vraisemblance

vraisemblance $L : \prod_{i=1}^{n} p(x_i)^{y_i} (1-p(x_i))^{1-y_i}$

Mais au lieu de maximiser L, on maximise $\ell = \ln(L) = \ln(\text{vraisemblance})$ pour des raisons computationnelles, car en introduisant la fonction logarithmique, on transforme le produit en une somme, ce qui simplifie les dérivations

$$\ell = \ln(L) = \ln\left[\prod_{i=1}^{n} p(x_i)^{y_i} (1-p(x_i))^{1-y_i}\right] = \sum_{i=1}^{n} \left\{ y_i \ln[p(x_i)] + (1-y_i)\ln[1-p(x_i)] \right\}$$

or $p[x_i] = \dfrac{1}{1+e^{-(\beta_0+\beta_1 x_i)}}$ et $\ln[p(x_i)] - \ln[1-p(x_i)] = \beta_0 + \beta_1 x_i$

donc $\ell = \sum_{i=1}^{n} \left[y_i \ln[p(x_i)] + \ln[1-p(x_i)] - y_i \ln[1-p(x_i)] \right]$

$$\ell = \sum_{i=1}^{n} \left[y_i [\ln(p(x_i)) - \ln(1-p(x_i))] + \ln(1-p(x_i)) \right]$$

$$\ell = \sum_{i=1}^{n} \left[y_i (\beta_0 + \beta_1 x_i) + \ln\left[1 - \frac{1}{1+e^{-(\beta_0+\beta_1 x_i)}}\right] \right]$$

$$= \sum_{i=1}^{n} \left[y_i (\beta_0 + \beta_1 x_i) + \ln\left[\frac{e^{-(\beta_0+\beta_1 x_i)}}{1+e^{-(\beta_0+\beta_1 x_i)}}\right] \right]$$

$$= \sum_{i=1}^{n} \left[y_i (\beta_0 + \beta_1 x_i) + \ln\left[e^{-(\beta_0+\beta_1 x_i)}\right] - \ln\left[1+e^{-(\beta_0+\beta_1 x_i)}\right] \right]$$

$$= \sum_{i=1}^{n} \left[y_i (\beta_0 + \beta_1 x_i) - (\beta_0 + \beta_1 x_i) - \ln\left[1+e^{-(\beta_0+\beta_1 x_i)}\right] \right]$$

$$\ell = -\sum_{i=1}^{n} \left[(1-y_i)(\beta_0 + \beta_1 x_i) + \ln\left[1+e^{-(\beta_0+\beta_1 x_i)}\right] \right]$$

Alors, il faut choisir $\hat{\beta}_0$ et $\hat{\beta}_1$ de façon à maximiser ℓ ou minimiser $-\ell$

donc $\frac{d\ell}{d\hat{\beta}_0} = 0 \Rightarrow \sum_{i=1}^{n} \left[(1-y_i) + \frac{-e^{-(\hat{\beta}_0+\hat{\beta}_1 x_i)}}{1+e^{-(\hat{\beta}_0+\hat{\beta}_1 x_i)}} \right] = 0$

$$\frac{d\ell}{d\hat{\beta}_1} = 0 \Rightarrow \sum_{i=1}^{n} \left[(1-y_i)x_i + \frac{-x_i e^{-(\hat{\beta}_0+\hat{\beta}_1 x_i)}}{1+e^{-(\hat{\beta}_0+\hat{\beta}_1 x_i)}} \right] = 0$$

On déduit $\hat{\beta}_0$ et $\hat{\beta}_1$ en résolvant ce système à deux équations

6) Après le calcul des estimateurs $\hat{\beta}_0$ et $\hat{\beta}_1$ (à partir des données d'apprentissage) le modèle calcule la probabilité estimée $\hat{y}$ du nouvel individu x_i

$$\hat{y}_i = \frac{1}{1+e^{-(\hat{\beta}_0+\hat{\beta}_1 x_i)}}$$

À partir de $\hat{y}$, et d'un seuil prédéfini (généralement 0,5), on décide la classe finale : si $\hat{y} \geq$ seuil $\Rightarrow$ la classe prédite est 1

si $\hat{y} <$ seuil $\Rightarrow$ la classe prédite est 0

partie-iv

November 29, 2023

1 TP - Réseau de Neurones

By: Omar ALLOUCH - Mathieu SROUR

[339]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.metrics import confusion_matrix, roc_curve, roc_auc_score,␣
↪classification_report
```

[340]:
```
data = pd.read_csv('HD_Complete_Data.csv', sep=';')
```

1.1 Exploratory Data Analysis

[341]:
```
data.describe().T
```

[341]:

	count	mean	std	min	25%	50%	75%	max
Age	918.0	53.510893	9.432617	28.0	47.00	54.0	60.0	77.0
RestingBP	918.0	132.396514	18.514154	0.0	120.00	130.0	140.0	200.0
Cholesterol	918.0	198.799564	109.384145	0.0	173.25	223.0	267.0	603.0
FastingBS	918.0	0.233115	0.423046	0.0	0.00	0.0	0.0	1.0
MaxHR	918.0	136.809368	25.460334	60.0	120.00	138.0	156.0	202.0
Oldpeak	918.0	0.887364	1.066570	-2.6	0.00	0.6	1.5	6.2
HeartDisease	918.0	0.553377	0.497414	0.0	0.00	1.0	1.0	1.0

[342]:
```
categorical_variables = ['Sexe', 'ChestPainType', 'RestingECG',␣
↪'ExerciseAngina', 'ST_Slope', 'FastingBS', 'HeartDisease']
data[categorical_variables] = data[categorical_variables].astype('category')
```

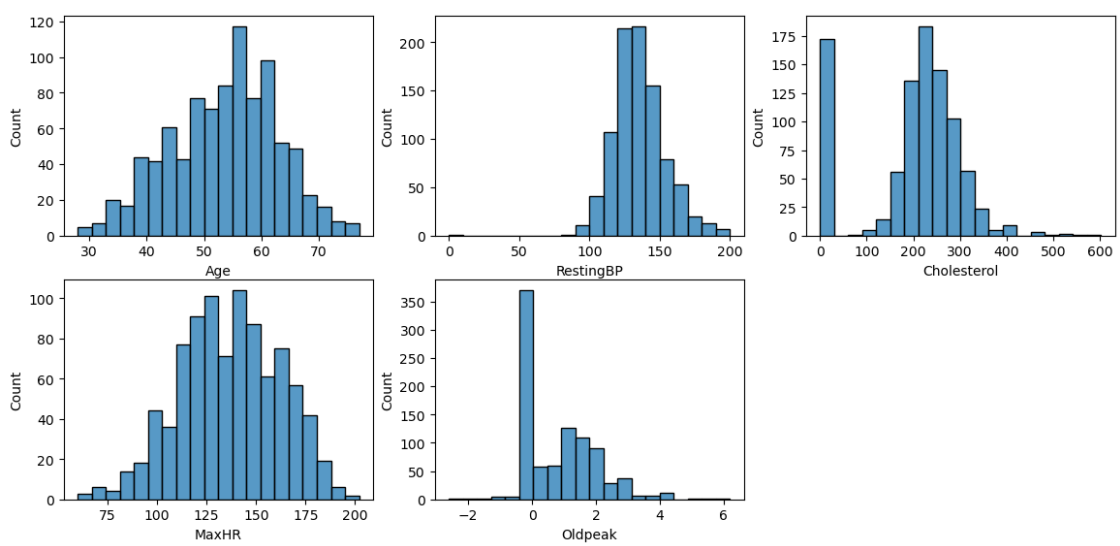
[343]:
```
data.isnull().sum()
```

[343]:
```
Age               0
Sexe              0
ChestPainType     0
RestingBP         0
Cholesterol       0
FastingBS         0
RestingECG        0
MaxHR             0
ExerciseAngina    0
Oldpeak           0
ST_Slope          0
HeartDisease      0
dtype: int64
```

No null values in the data.

Distribution of numerical features

[344]:
```
numerical_data = data.select_dtypes(include=['int64', 'float64'])
plt.figure(figsize=(14, 10))
for i, col in enumerate(numerical_data.columns):
    plt.subplot(3, 3, i+1)
    sns.histplot(data=numerical_data, x=col, bins=20)
```

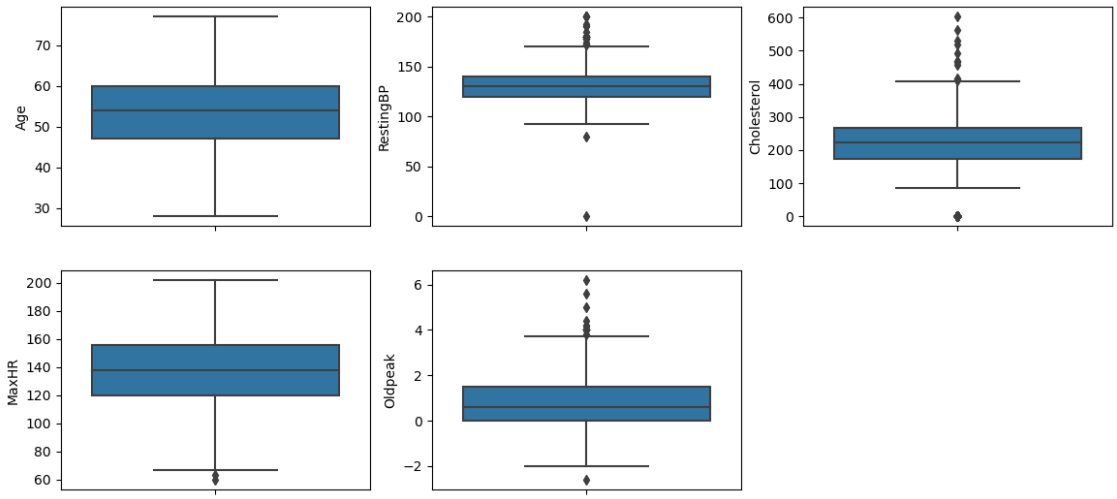


- **Oldpeak** distribution is right-skewed.
- **Cholesterol** has a bimodal distribution.

Boxplots

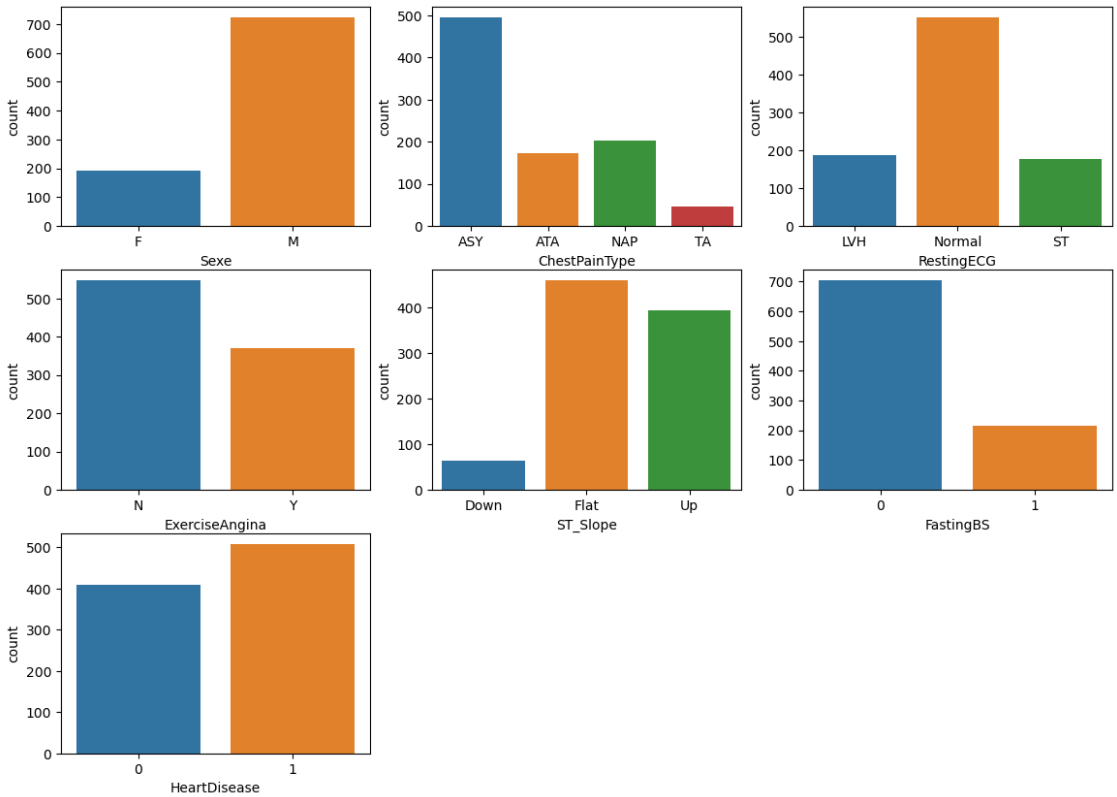
[345]:
```
plt.figure(figsize=(14, 10))
for i, col in enumerate(numerical_data.columns):
```

```
    plt.subplot(3, 3, i+1)
    sns.boxplot(y=col, data=numerical_data)
```



Categorical variables

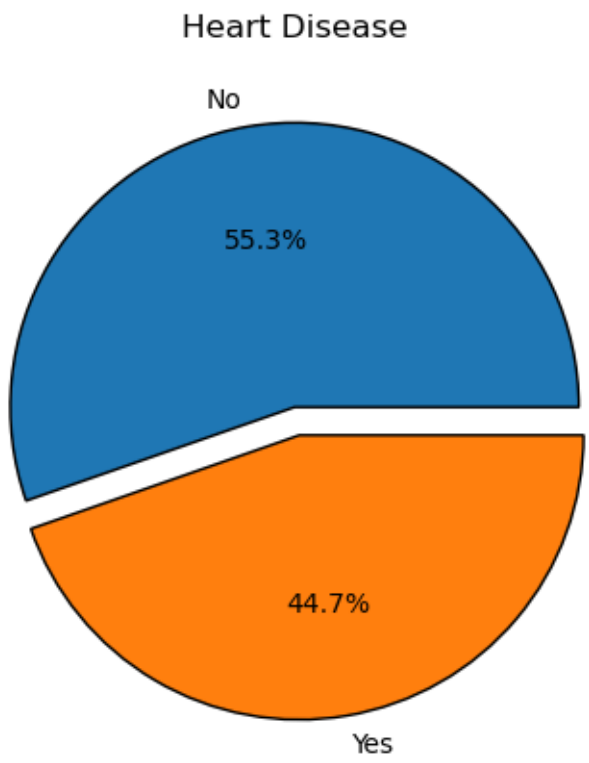
[346]:
```
plt.figure(figsize=(14, 10))
for i, col in enumerate(categorical_variables):
    plt.subplot(3, 3, i+1)
    sns.countplot(x=col, data=data)
```



Target variable

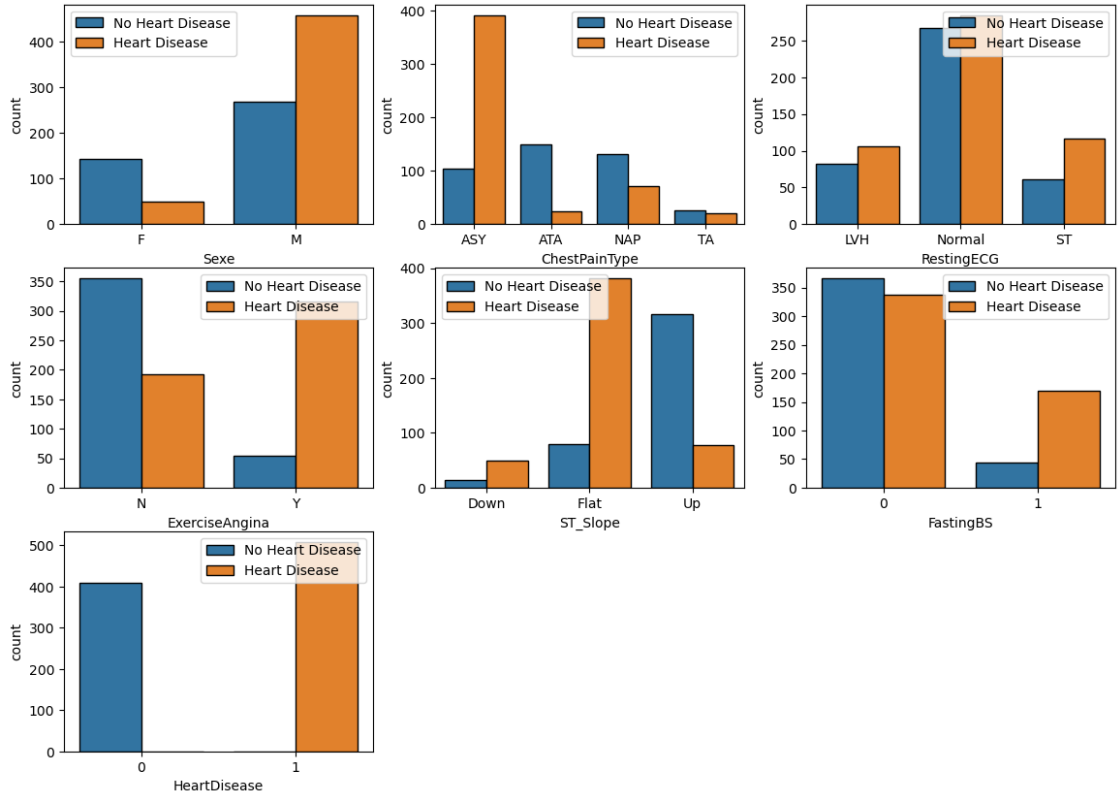
[347]:
```
plt.pie(data['HeartDisease'].value_counts(), labels=['No', 'Yes'], autopct='%1.
↪1f%%', explode=[0, 0.1], wedgeprops={'edgecolor': 'black'})
plt.title('Heart Disease')
```

[347]: Text(0.5, 1.0, 'Heart Disease')



Categorical variables vs target variable

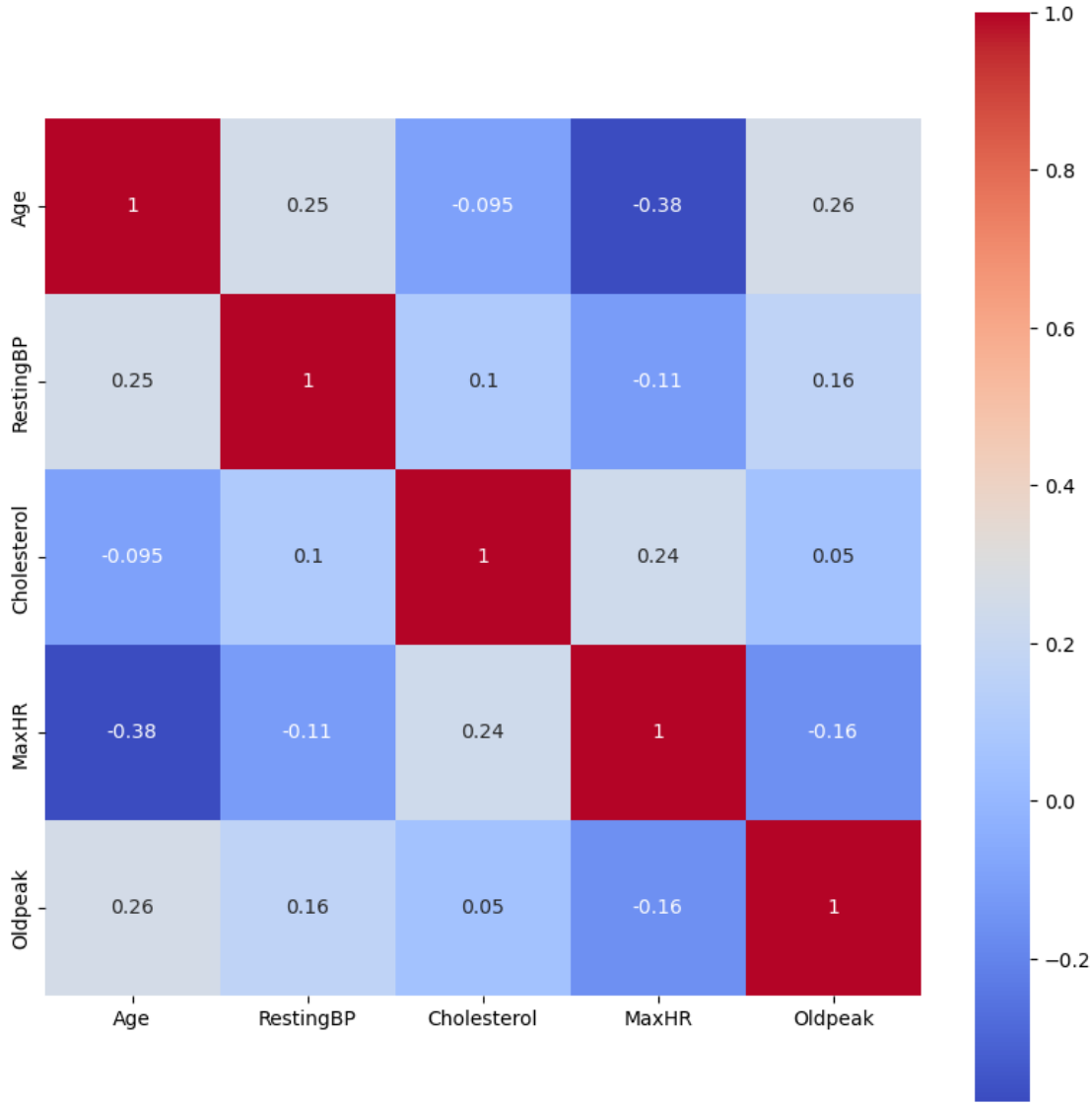
[348]:
```
# Categorical variables vs target variable
plt.figure(figsize=(14, 10))
for i, col in enumerate(categorical_variables):
    plt.subplot(3, 3, i+1)
    sns.countplot(x=col, hue='HeartDisease', data=data, edgecolor='black')
    plt.legend(['No Heart Disease', 'Heart Disease'])
```



- **Males** population has more *heart disease* individuals than *no heart disease*. The opposite is for **Females**.
- **ASY** *chest pain type* is a strong indicator of a *heart disease*.
- **ExerciseAngina** greatly suggests a *heart disease*.
- In **ST_Slope**, a **flat** slope indicates a *heart disease*, same for **down**.
- **RestingECG** and **FastingBS** are not so clear.

Correlations

[349]:
```
corr = numerical_data.corr()
plt.figure(figsize=(10, 10))
sns.heatmap(corr, annot=True, square=True, cmap='coolwarm')
plt.show()
```



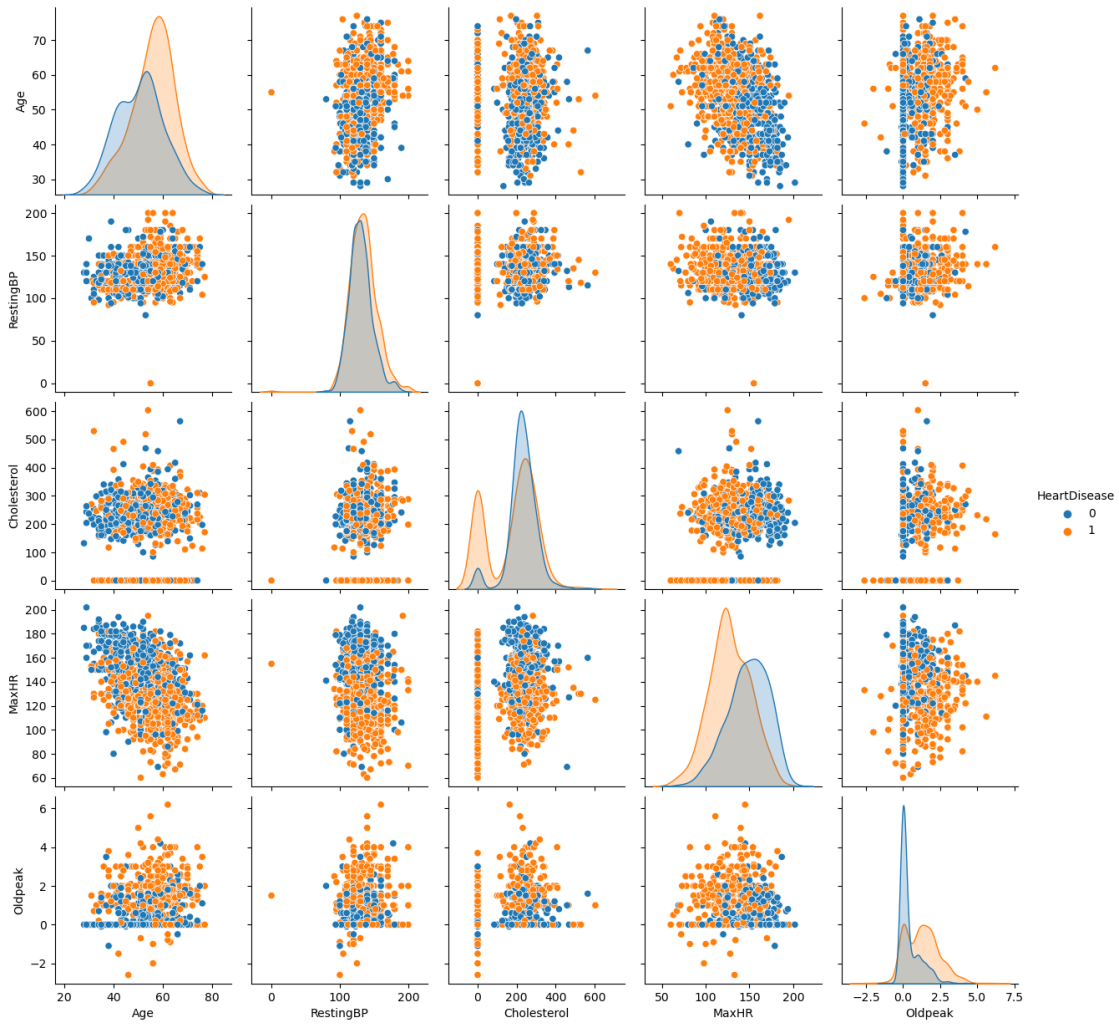
No major correlations, we will keep all the numerical variables.

Pairplot of numerical variables

[350]:
```
sns.pairplot(data, hue='HeartDisease')
```

```
/home/omar/anaconda3/lib/python3.11/site-packages/seaborn/axisgrid.py:118:
UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

[350]: <seaborn.axisgrid.PairGrid at 0x7fa313bfa4d0>



From the various plots we obtained, we can distinguish some outliers, and we shall handle those.

Furthermore, the scale of variables is not the same which would negatively affect the performance, so we will opt to standardize those.

1.2 Data Preprocessing

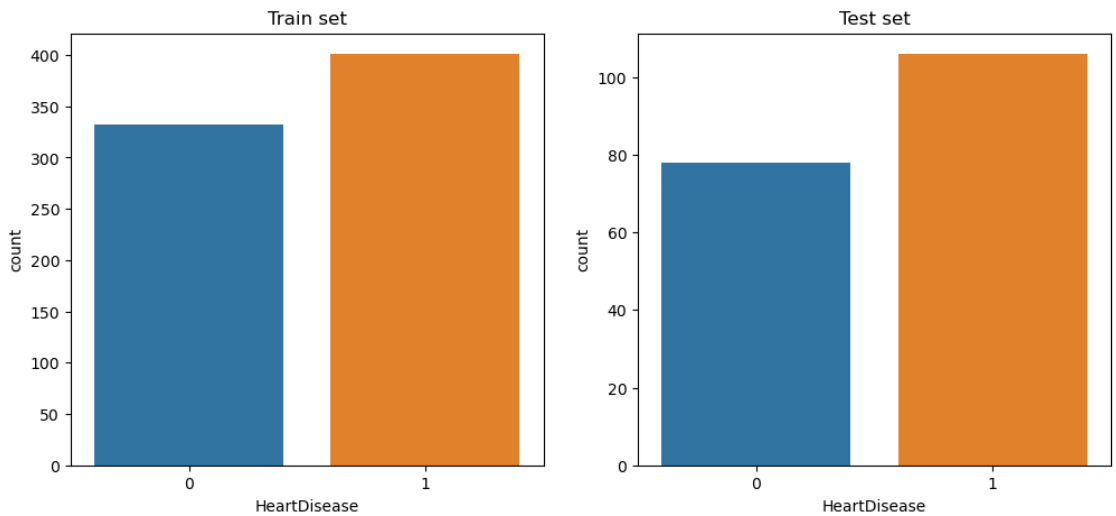
Removing outliers

[351]:
```
data = data[data['RestingBP'] > 50]
```

[352]:
```
X, y = data.drop('HeartDisease', axis=1), data['HeartDisease']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=1234)

print(f'X_train shape: {X_train.shape}')
print(f'X_test shape: {X_test.shape}')
```

```
X_train shape: (733, 11)
X_test shape: (184, 11)
```

[353]:
```
# Plot the distribution of the target variable for train and test sets
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
sns.countplot(x=y_train)
plt.title('Train set')
plt.subplot(1, 2, 2)
sns.countplot(x=y_test)
plt.title('Test set')
plt.show()
```



Notice how we kept the same proportion of the target variable in both train and test sets.

Now we scale our variables for the reasons mentioned above.

[354]:
```
scaler = StandardScaler()
min_max_scaler = MinMaxScaler()

numerical_data = X_train.select_dtypes(include=['int64', 'float64']).
 ↪drop('Oldpeak', axis=1)
X_train[numerical_data.columns] = scaler.fit_transform(numerical_data)
X_train['Oldpeak'] = min_max_scaler.fit_transform(X_train[['Oldpeak']])

numerical_data = X_test.select_dtypes(include=['int64', 'float64']).
 ↪drop('Oldpeak', axis=1)
X_test[numerical_data.columns] = scaler.fit_transform(numerical_data)
X_test['Oldpeak'] = min_max_scaler.fit_transform(X_test[['Oldpeak']])
```

We'll one-hot encode our categorical variables, so as to not give any unwanted importance for any

category over the others.

[355]:
```
# One-hot encoding
categorical_variables = ['Sexe', 'ChestPainType', 'RestingECG',␣
↪'ExerciseAngina', 'ST_Slope', 'FastingBS']

one_hot_encoded = pd.get_dummies(X_train[categorical_variables],␣
↪prefix=categorical_variables)
X_train = pd.concat([X_train, one_hot_encoded], axis=1)
X_train.drop(columns=categorical_variables, axis=1, inplace=True)
X_train = np.asarray(X_train).astype('float32')

one_hot_encoded = pd.get_dummies(X_test[categorical_variables],␣
↪prefix=categorical_variables)
X_test = pd.concat([X_test, one_hot_encoded], axis=1)
X_test.drop(columns=categorical_variables, axis=1, inplace=True)
X_test = np.asarray(X_test).astype('float32')

print(f'X_train shape: {X_train.shape}')
print(f'X_test shape: {X_test.shape}')
```

```
X_train shape: (733, 21)
X_test shape: (184, 21)
```

The number of columns increased from 11 to 21 (variables), which is inline with the number of categories of different variables.

1.3 Model training

[356]:
```
# Set the seed for reproducibility
tf.random.set_seed(1234)
np.random.seed(1234)

# Construct the neural network
model = tf.keras.Sequential([
    tf.keras.layers.Dense(X_train.shape[1], activation='relu',␣
↪input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid'),
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',␣
↪metrics=['accuracy'])

model.summary()
```

```
Model: "sequential_23"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_112 (Dense)           (None, 21)                462

 dense_113 (Dense)           (None, 32)                704

 dense_114 (Dense)           (None, 64)                2112

 dense_115 (Dense)           (None, 32)                2080

 dense_116 (Dense)           (None, 1)                 33

=================================================================
Total params: 5,391
Trainable params: 5,391
Non-trainable params: 0
_________________________________________________________________
```

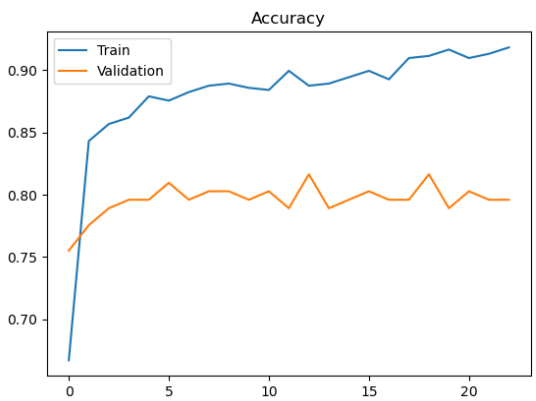
We will be adding an `early_stopping` criteria to prevent the model from overfitting. Another way would be to add some `Dropout` layers, however, by doing some trial and error the early stopping seemed to do the job and produce better results.

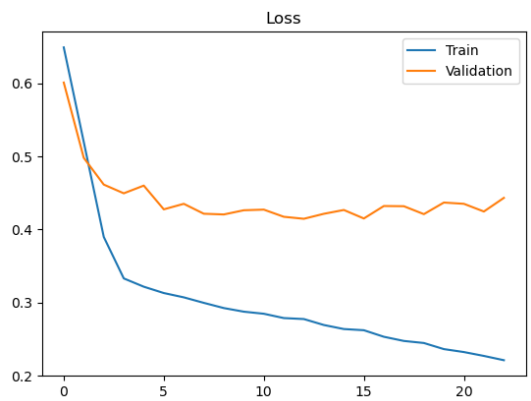
[357]:
```
# Early stopping
early_stopping = tf.keras.callbacks.EarlyStopping(patience=10,␣
 ↪restore_best_weights=True)

# Fit the model
history = model.fit(X_train, y_train, validation_split=0.2, epochs=100,␣
 ↪callbacks=[early_stopping], use_multiprocessing=True)

# Plot the learning curves
plt.figure(figsize=(14, 10))
plt.subplot(2, 2, 1)
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Validation')
plt.title('Accuracy')
plt.legend()
plt.subplot(2, 2, 2)
plt.plot(history.history['loss'], label='Train')
plt.plot(history.history['val_loss'], label='Validation')
plt.title('Loss')
plt.legend()
plt.show()
```

```
Epoch 1/100
19/19 [==============================] - 1s 27ms/step - loss: 0.6491 - accuracy:
```

```
0.6672 - val_loss: 0.6010 - val_accuracy: 0.7551
Epoch 2/100
19/19 [==============================] - 0s 16ms/step - loss: 0.5198 - accuracy:
0.8430 - val_loss: 0.4980 - val_accuracy: 0.7755
Epoch 3/100
19/19 [==============================] - 0s 12ms/step - loss: 0.3898 - accuracy:
0.8567 - val_loss: 0.4615 - val_accuracy: 0.7891
Epoch 4/100
19/19 [==============================] - 0s 10ms/step - loss: 0.3331 - accuracy:
0.8618 - val_loss: 0.4495 - val_accuracy: 0.7959
Epoch 5/100
19/19 [==============================] - 0s 15ms/step - loss: 0.3217 - accuracy:
0.8788 - val_loss: 0.4601 - val_accuracy: 0.7959
Epoch 6/100
19/19 [==============================] - 0s 13ms/step - loss: 0.3131 - accuracy:
0.8754 - val_loss: 0.4276 - val_accuracy: 0.8095
Epoch 7/100
19/19 [==============================] - 0s 11ms/step - loss: 0.3072 - accuracy:
0.8823 - val_loss: 0.4351 - val_accuracy: 0.7959
Epoch 8/100
19/19 [==============================] - 0s 18ms/step - loss: 0.2996 - accuracy:
0.8874 - val_loss: 0.4216 - val_accuracy: 0.8027
Epoch 9/100
19/19 [==============================] - 0s 17ms/step - loss: 0.2925 - accuracy:
0.8891 - val_loss: 0.4206 - val_accuracy: 0.8027
Epoch 10/100
19/19 [==============================] - 0s 11ms/step - loss: 0.2876 - accuracy:
0.8857 - val_loss: 0.4263 - val_accuracy: 0.7959
Epoch 11/100
19/19 [==============================] - 0s 13ms/step - loss: 0.2847 - accuracy:
0.8840 - val_loss: 0.4272 - val_accuracy: 0.8027
Epoch 12/100
19/19 [==============================] - 0s 12ms/step - loss: 0.2788 - accuracy:
0.8993 - val_loss: 0.4174 - val_accuracy: 0.7891
Epoch 13/100
19/19 [==============================] - 0s 16ms/step - loss: 0.2775 - accuracy:
0.8874 - val_loss: 0.4147 - val_accuracy: 0.8163
Epoch 14/100
19/19 [==============================] - 0s 24ms/step - loss: 0.2693 - accuracy:
0.8891 - val_loss: 0.4215 - val_accuracy: 0.7891
Epoch 15/100
19/19 [==============================] - 0s 12ms/step - loss: 0.2638 - accuracy:
0.8942 - val_loss: 0.4267 - val_accuracy: 0.7959
Epoch 16/100
19/19 [==============================] - 0s 15ms/step - loss: 0.2622 - accuracy:
0.8993 - val_loss: 0.4151 - val_accuracy: 0.8027
Epoch 17/100
19/19 [==============================] - 0s 12ms/step - loss: 0.2532 - accuracy:
```

```
0.8925 - val_loss: 0.4321 - val_accuracy: 0.7959
Epoch 18/100
19/19 [==============================] - 0s 10ms/step - loss: 0.2475 - accuracy:
0.9096 - val_loss: 0.4319 - val_accuracy: 0.7959
Epoch 19/100
19/19 [==============================] - 0s 15ms/step - loss: 0.2447 - accuracy:
0.9113 - val_loss: 0.4210 - val_accuracy: 0.8163
Epoch 20/100
19/19 [==============================] - 0s 14ms/step - loss: 0.2364 - accuracy:
0.9164 - val_loss: 0.4369 - val_accuracy: 0.7891
Epoch 21/100
19/19 [==============================] - 0s 11ms/step - loss: 0.2323 - accuracy:
0.9096 - val_loss: 0.4352 - val_accuracy: 0.8027
Epoch 22/100
19/19 [==============================] - 0s 12ms/step - loss: 0.2270 - accuracy:
0.9130 - val_loss: 0.4246 - val_accuracy: 0.7959
Epoch 23/100
19/19 [==============================] - 0s 19ms/step - loss: 0.2212 - accuracy:
0.9181 - val_loss: 0.4433 - val_accuracy: 0.7959
```

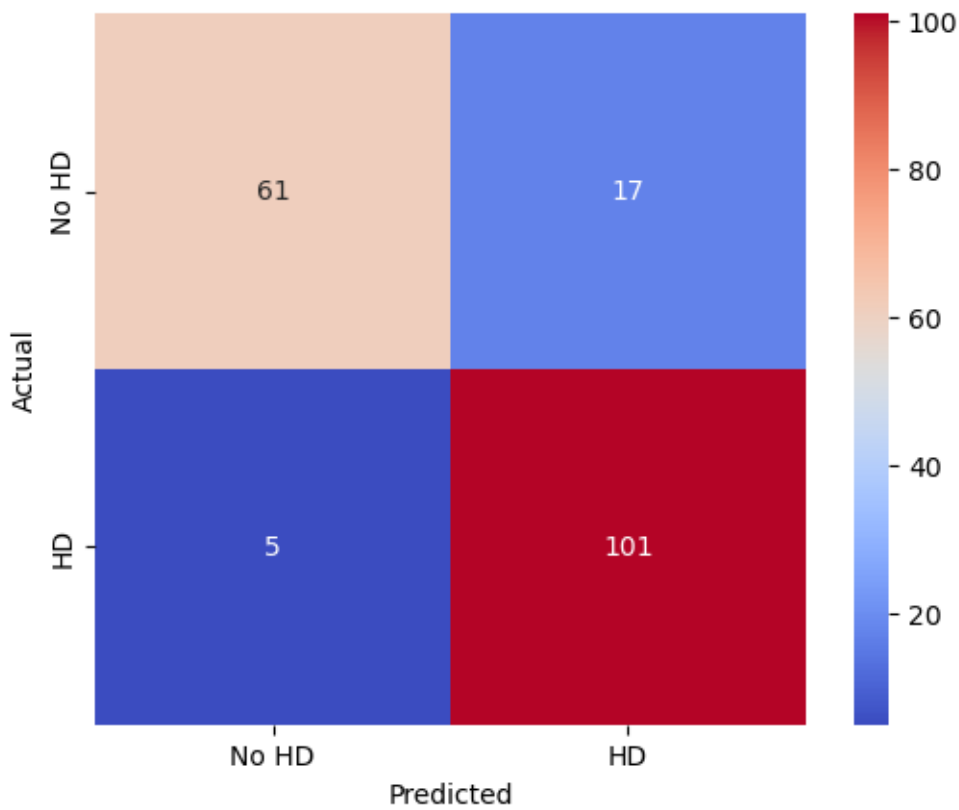




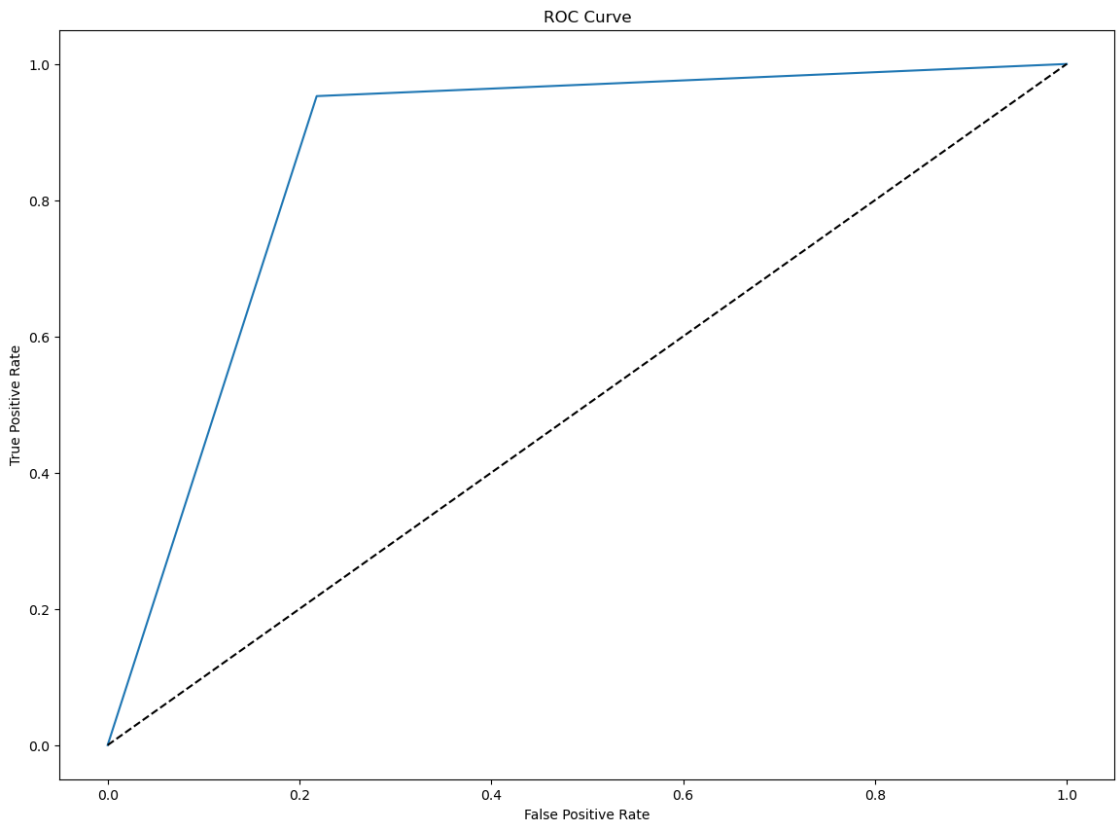
[358]:
```python
# Evaluate the model
model.evaluate(X_test, y_test)

# Make predictions
y_pred = model.predict(X_test)
y_pred = y_pred > 0.5

print("Classification report:")
print(classification_report(y_test, y_pred))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
```

```
sns.heatmap(cm, annot=True, square=True, cmap='coolwarm', fmt='d',␣
 ↪xticklabels=['No HD', 'HD'], yticklabels=['No HD', 'HD'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

```
6/6 [==============================] - 0s 5ms/step - loss: 0.3454 - accuracy:
0.8804
6/6 [==============================] - 0s 3ms/step
Classification report:
              precision    recall  f1-score   support

           0       0.92      0.78      0.85        78
           1       0.86      0.95      0.90       106

    accuracy                           0.88       184
   macro avg       0.89      0.87      0.87       184
weighted avg       0.88      0.88      0.88       184
```



[359]:
```
# Plot the ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred)
plt.figure(figsize=(14, 10))
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()

# Compute the AUC score
print("AUC Score: ", roc_auc_score(y_test, y_pred))
```

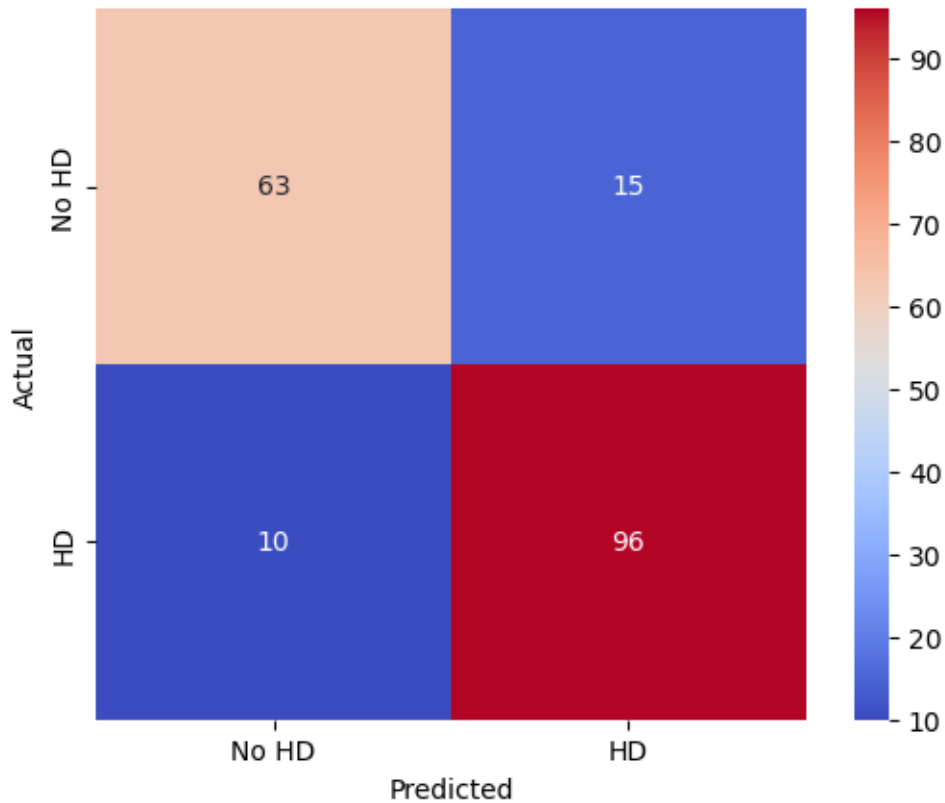



```
AUC Score:  0.8674407353652636
```

[361]:
```
# Logistic regression model
logistic_model = LogisticRegression()
logistic_model.fit(X_train, y_train)

# Make predictions
y_pred = logistic_model.predict(X_test)
```

```python
print("Classification report:")
print(classification_report(y_test, y_pred))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, square=True, cmap='coolwarm', fmt='d',␣
 ↪xticklabels=['No HD', 'HD'], yticklabels=['No HD', 'HD'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

# Compute the AUC score
print("AUC Score: ", roc_auc_score(y_test, y_pred))
```

```
Classification report:
              precision    recall  f1-score   support

           0       0.86      0.81      0.83        78
           1       0.86      0.91      0.88       106

    accuracy                           0.86       184
   macro avg       0.86      0.86      0.86       184
weighted avg       0.86      0.86      0.86       184
```




```
AUC Score:  0.8566763425253991
```

1.4 Results

Recap of results: - NN: accuracy = 88%, auc = 0.87 - Logistic regression: accuracy = 86%, auc = 0.8566

These 2 metrics may seem very close, however we should also take a look at the precision, recall, and f1-score which are better for the NN. Moreover, the NN had less false negatives, which is critical in this kind of applications. The major disadvantage is the huge number of parameters in the NN, and some would argue that the results are not worth it. Personally, I see that the NN is still better, and with hyper-parameter tuning we could come up with even better results, while always retaining a low false negative rate, which would allow us to rely more on the model for real-life predictions.

On another note, it could be interesting to look at the results of models such as: SVM (SVC), RandomForest, XGBoost…etc and compare the results.

problem-5-mathieu-and-omar

November 29, 2023

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense, Dropout, SimpleRNN, LSTM
from keras.callbacks import EarlyStopping
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import BatchNormalization
import pmdarima as pm
from pmdarima.model_selection import train_test_split

import warnings
warnings.filterwarnings("ignore")
```

```
WARNING:tensorflow:From C:\Users\lenovo\anaconda3\lib\site-
packages\keras\src\losses.py:2976: The name
tf.losses.sparse_softmax_cross_entropy is deprecated. Please use
tf.compat.v1.losses.sparse_softmax_cross_entropy instead.
```

0.1 Import the data

[3]:
```
data = pd.read_csv("Echantillons de données-20231125\\Danube ammonium level␣
↪Time Series.csv", sep = ";", index_col="Months")
data
```

[3]:

	Ammonium
Months	
01/01/1996	0,253333
01/02/1996	0,218889
01/03/1996	0,242
01/04/1996	0,134545
01/05/1996	0,113333
…	…

```
01/08/2017  0,031222
01/09/2017  0,045875
01/10/2017  0,032474
01/11/2017  0,047125
01/12/2017  0,056563

[264 rows x 1 columns]
```

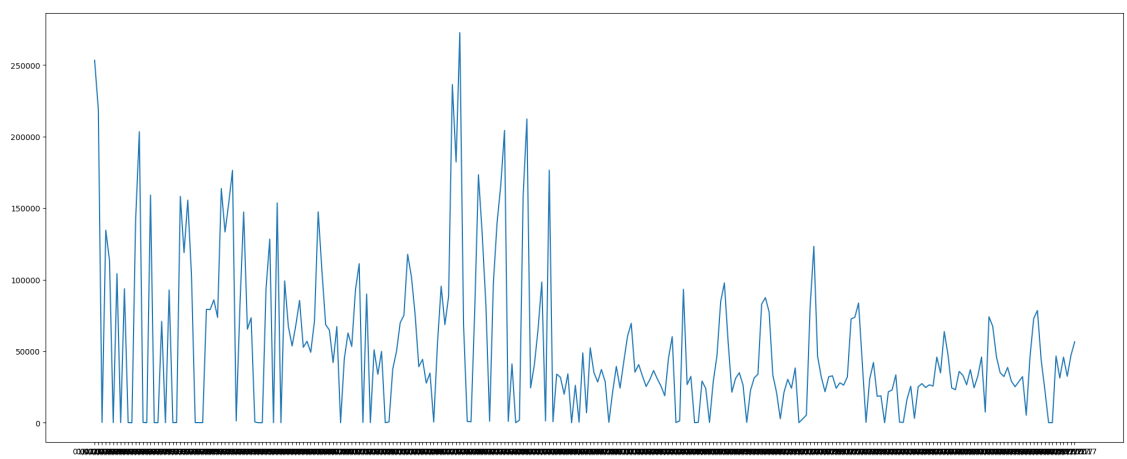
[4]:
```
data['Ammonium'] = data['Ammonium'].apply(lambda x: str(x).split(',')[-1])
data['Ammonium'] = data['Ammonium'].astype(int)
data
```

[4]:
```
            Ammonium
Months
01/01/1996    253333
01/02/1996    218889
01/03/1996       242
01/04/1996    134545
01/05/1996    113333
…               …
01/08/2017     31222
01/09/2017     45875
01/10/2017     32474
01/11/2017     47125
01/12/2017     56563

[264 rows x 1 columns]
```

0.2 Plot of the Time Series

[5]:
```
plt.figure(figsize=(25,10))
plt.plot(data['Ammonium'])
```

[5]:
```
[<matplotlib.lines.Line2D at 0x2c92745fee0>]
```



0.2.1 Split the data

[6]:
```
training_set = data.iloc[:252]
test_set = data.iloc[-13:]
```

0.2.2 Scaling the data

[7]:
```
sc = MinMaxScaler(feature_range=(0, 1))
training_set_scaled = sc.fit_transform(training_set)
test_set_scaled = sc.transform(test_set)
```

0.2.3 Préparation de la donnée

[8]:
```
#representation des donnees sous 2 colonnes: x qui represente les entree et y␣
↪qui repreente les sorties correspondantes
#La valeur de sortie correspondante a une entree est l'élément à l'étape␣
↪temporelle suivante.

def split_sequence(sequence, n_steps):
    X, y = list(), list()
    for i in range(len(sequence)):
        end_ix = i + n_steps
        if end_ix > len(sequence) - 1:
            break
        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
        X.append(seq_x)
        y.append(seq_y)
    return np.array(X), np.array(y)
```

[9]:
```
def plot_predictions(test, predicted):
    plt.plot(test, color="gray", label="Real")
    plt.plot(predicted, color="red", label="Predicted")
    plt.title("MasterCard Stock Price Prediction")
    plt.xlabel("Time")
    plt.ylabel("MasterCard Stock Price")
    plt.legend()
    plt.show()
```

[10]:
```
def return_rmse(test, predicted):
    rmse = np.sqrt(mean_squared_error(test, predicted))
    print("The root mean squared error is {:.2f}.".format(rmse))
```

[11]:
```
n_steps = 1
features = 1
X_train, y_train = split_sequence(training_set_scaled, n_steps)
```

[12]:
```
# Reshaping X_train for model
X_train = X_train.reshape(X_train.shape[0],X_train.shape[1],features)
```

[13]:
```
X_test, y_test = split_sequence(test_set_scaled, 1)
# Reshaping X_test for model
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)
```

0.3 Model 1

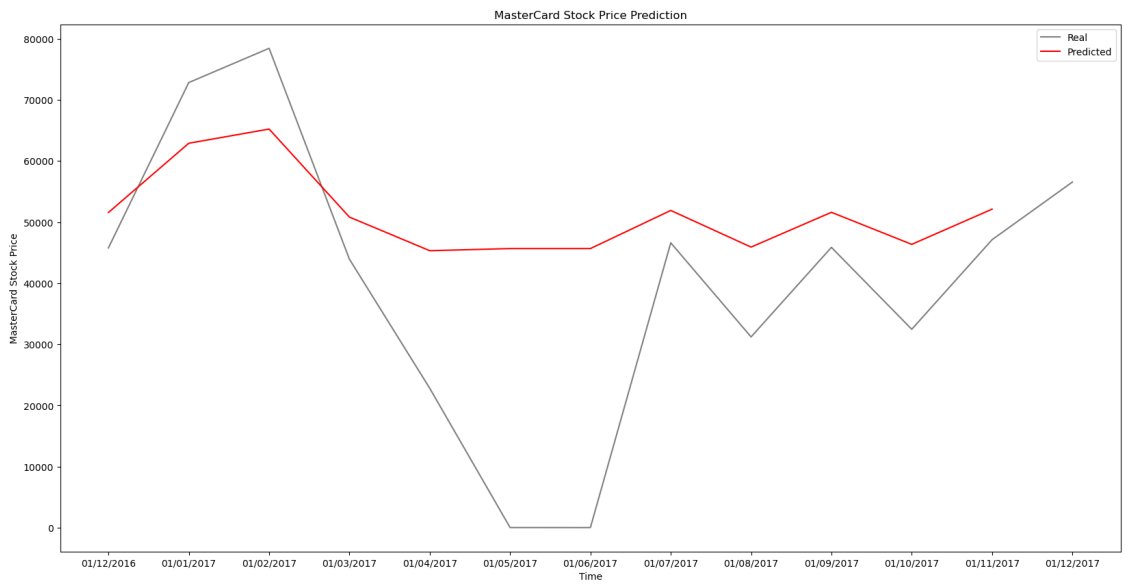
[14]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from kerastuner.tuners import RandomSearch

# Define your RNN model
def build_model(hp):
    model = Sequential()

    model.add(layers.SimpleRNN(units=hp.Int('units', min_value=32,␣
 ↪max_value=512, step=32),
                               input_shape=(X_train.shape[1], X_train.shape[2]),
                               activation='relu'))

    model.add(layers.Dense(units=hp.Int('dense_units', min_value=32,␣
 ↪max_value=256, step=32),
                           activation='relu'))

    model.add(layers.Dropout(rate=hp.Float('dropout_rate', min_value=0.1,␣
 ↪max_value=0.5, step=0.1)))

    model.add(layers.Dense(units=hp.Int('dense_units_2', min_value=16,␣
 ↪max_value=128, step=16),
                           activation='relu'))

    model.add(layers.Dense(1, activation='linear'))

    # Compile the model with a learning rate hyperparameter
    model.compile(optimizer=keras.optimizers.Adam(learning_rate=hp.
 ↪Float('learning_rate', min_value=1e-4, max_value=1e-2, sampling='log')),
                  loss='mean_squared_error',
                  metrics=['mae'])
```

```
    return model

tuner = RandomSearch(
    build_model,
    objective='val_mae',
    max_trials=20,  # Number of hyperparameter combinations to try
    directory='hyperparameter_tuning',
    project_name='simple_rnn_tuning'
)

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.
 ↪2)

tuner.search(x=X_train, y=y_train, epochs=10, validation_data=(X_val, y_val))

best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
print(f"Best Hyperparameters: {best_hps}")

final_model = tuner.hypermodel.build(best_hps)

final_model.fit(X_train, y_train, epochs=20, validation_data=(X_val, y_val))

predicted_price3 = final_model.predict(X_test)
predicted_price3 = sc.inverse_transform(predicted_price3)

plt.figure(figsize=(20,10))
plot_predictions(test_set,predicted_price3)
return_rmse(test_set[:-1],predicted_price3)
```

```
Reloading Tuner from hyperparameter_tuning\simple_rnn_tuning\tuner0.json
Best Hyperparameters:
<keras_tuner.src.engine.hyperparameters.hyperparameters.HyperParameters object
at 0x000002C92761FD00>
WARNING:tensorflow:From C:\Users\lenovo\anaconda3\lib\site-
packages\keras\src\backend.py:873: The name tf.get_default_graph is deprecated.
Please use tf.compat.v1.get_default_graph instead.

Epoch 1/20
WARNING:tensorflow:From C:\Users\lenovo\anaconda3\lib\site-
packages\keras\src\utils\tf_utils.py:492: The name tf.ragged.RaggedTensorValue
is deprecated. Please use tf.compat.v1.ragged.RaggedTensorValue instead.

WARNING:tensorflow:From C:\Users\lenovo\anaconda3\lib\site-
packages\keras\src\engine\base_layer_utils.py:384: The name
tf.executing_eagerly_outside_functions is deprecated. Please use
tf.compat.v1.executing_eagerly_outside_functions instead.
```

```
7/7 [==============================] - 1s 36ms/step - loss: 0.1027 - mae: 0.2379
- val_loss: 0.0155 - val_mae: 0.1043
Epoch 2/20
7/7 [==============================] - 0s 7ms/step - loss: 0.0771 - mae: 0.1983
- val_loss: 0.0094 - val_mae: 0.0795
Epoch 3/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0592 - mae: 0.1727
- val_loss: 0.0056 - val_mae: 0.0563
Epoch 4/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0471 - mae: 0.1552
- val_loss: 0.0042 - val_mae: 0.0457
Epoch 5/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0404 - mae: 0.1457
- val_loss: 0.0052 - val_mae: 0.0579
Epoch 6/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0390 - mae: 0.1475
- val_loss: 0.0069 - val_mae: 0.0703
Epoch 7/20
7/7 [==============================] - 0s 7ms/step - loss: 0.0362 - mae: 0.1446
- val_loss: 0.0082 - val_mae: 0.0782
Epoch 8/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0376 - mae: 0.1498
- val_loss: 0.0082 - val_mae: 0.0785
Epoch 9/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0399 - mae: 0.1525
- val_loss: 0.0075 - val_mae: 0.0744
Epoch 10/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0376 - mae: 0.1474
- val_loss: 0.0064 - val_mae: 0.0681
Epoch 11/20
7/7 [==============================] - 0s 7ms/step - loss: 0.0381 - mae: 0.1464
- val_loss: 0.0062 - val_mae: 0.0666
Epoch 12/20
7/7 [==============================] - 0s 7ms/step - loss: 0.0387 - mae: 0.1487
- val_loss: 0.0062 - val_mae: 0.0666
Epoch 13/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0383 - mae: 0.1475
- val_loss: 0.0067 - val_mae: 0.0703
Epoch 14/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0372 - mae: 0.1479
- val_loss: 0.0075 - val_mae: 0.0750
Epoch 15/20
7/7 [==============================] - 0s 7ms/step - loss: 0.0391 - mae: 0.1493
- val_loss: 0.0074 - val_mae: 0.0742
Epoch 16/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0387 - mae: 0.1494
- val_loss: 0.0075 - val_mae: 0.0748
Epoch 17/20
```

```
7/7 [==============================] - 0s 7ms/step - loss: 0.0399 - mae: 0.1493
- val_loss: 0.0072 - val_mae: 0.0727
Epoch 18/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0367 - mae: 0.1450
- val_loss: 0.0074 - val_mae: 0.0740
Epoch 19/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0376 - mae: 0.1492
- val_loss: 0.0075 - val_mae: 0.0747
Epoch 20/20
7/7 [==============================] - 0s 6ms/step - loss: 0.0364 - mae: 0.1455
- val_loss: 0.0077 - val_mae: 0.0757
1/1 [==============================] - 0s 143ms/step
```




```
The root mean squared error is 21451.34.
```

0.4 Model 2

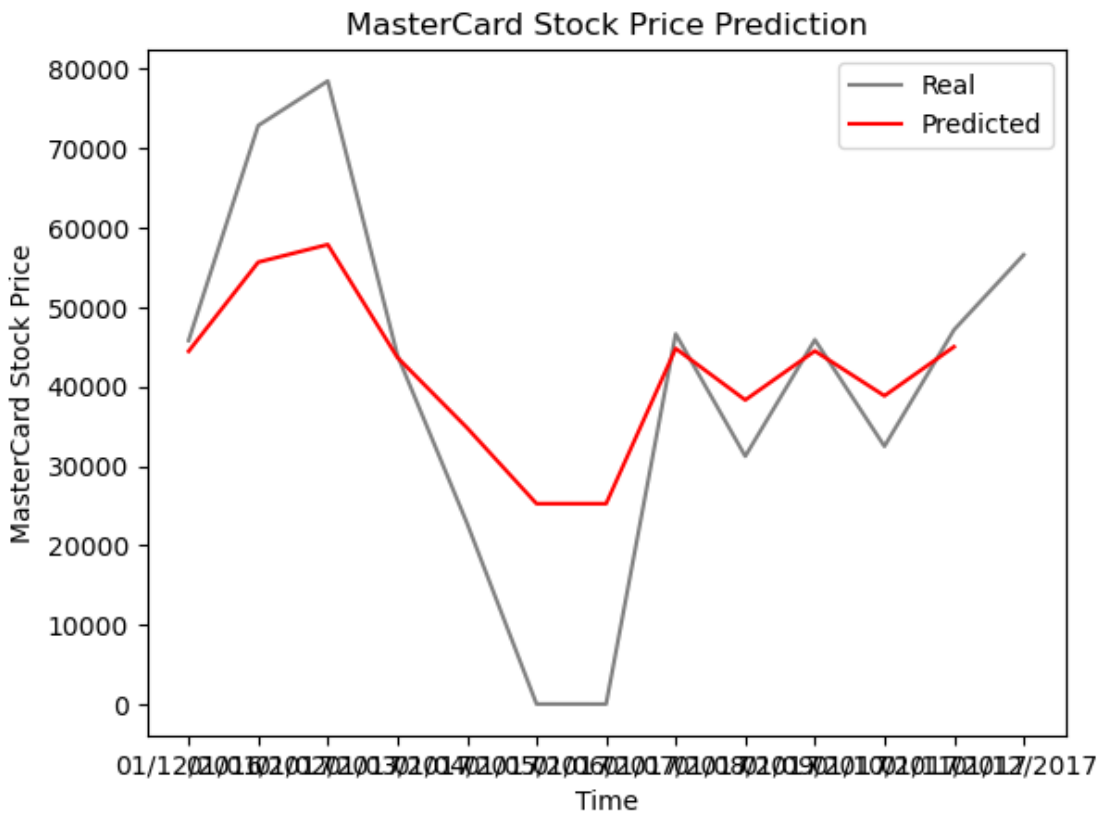
Afte many trials and changing the number of layers, neurons per layer, dropout value, learning rate ...., This was the best model

[46]:
```
model = Sequential()
model.add(SimpleRNN(256, input_shape = (X_train.shape[1], X_train.shape[2]),␣
 ↪activation = 'relu'))
model.add(Dropout(0.4))
model.add(Dense(units=1, activation = 'linear'))
model.add(Dropout(0.1))

optimizer = Adam(learning_rate=0.001)
early = EarlyStopping(monitor = "val_loss", mode = "min", patience = 6)
```

```
callbacks_list = [early]
model.compile(optimizer=optimizer, loss='mean_squared_error')
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val,␣
 ↪y_val))

history = model.fit(X_train, y_train,
    epochs=50, batch_size=20, verbose=2,
    validation_split = 0.20,
    callbacks = callbacks_list)

predicted_price2 = model.predict(X_test)
predicted_price2 = sc.inverse_transform(predicted_price2)

plt.figure(figsize=(20, 10))
plot_predictions(test_set,predicted_price2)

return_rmse(test_set[:-1],predicted_price2)
```

```
Epoch 1/10
7/7 [==============================] - 3s 105ms/step - loss: 0.0652 - val_loss:
0.0613
Epoch 2/10
7/7 [==============================] - 0s 20ms/step - loss: 0.0449 - val_loss:
0.0449
Epoch 3/10
7/7 [==============================] - 0s 19ms/step - loss: 0.0402 - val_loss:
0.0378
Epoch 4/10
7/7 [==============================] - 0s 19ms/step - loss: 0.0360 - val_loss:
0.0362
Epoch 5/10
7/7 [==============================] - 0s 19ms/step - loss: 0.0366 - val_loss:
0.0360
Epoch 6/10
7/7 [==============================] - 0s 19ms/step - loss: 0.0381 - val_loss:
0.0362
Epoch 7/10
7/7 [==============================] - 0s 19ms/step - loss: 0.0351 - val_loss:
0.0369
Epoch 8/10
7/7 [==============================] - 0s 19ms/step - loss: 0.0402 - val_loss:
0.0378
Epoch 9/10
7/7 [==============================] - 0s 19ms/step - loss: 0.0339 - val_loss:
0.0380
Epoch 10/10
7/7 [==============================] - 0s 20ms/step - loss: 0.0341 - val_loss:
```

```
0.0383
Epoch 1/50
8/8 - 0s - loss: 0.0350 - val_loss: 0.0422 - 211ms/epoch - 26ms/step
Epoch 2/50
8/8 - 0s - loss: 0.0331 - val_loss: 0.0425 - 108ms/epoch - 14ms/step
Epoch 3/50
8/8 - 0s - loss: 0.0392 - val_loss: 0.0427 - 116ms/epoch - 14ms/step
Epoch 4/50
8/8 - 0s - loss: 0.0334 - val_loss: 0.0429 - 109ms/epoch - 14ms/step
Epoch 5/50
8/8 - 0s - loss: 0.0321 - val_loss: 0.0431 - 120ms/epoch - 15ms/step
Epoch 6/50
8/8 - 0s - loss: 0.0350 - val_loss: 0.0435 - 119ms/epoch - 15ms/step
Epoch 7/50
8/8 - 0s - loss: 0.0305 - val_loss: 0.0439 - 117ms/epoch - 15ms/step
1/1 [==============================] - 0s 375ms/step
```




```
The root mean squared error is 13652.41.
```

1 ARIMA

We used the auto_arima function, It's a function already implemented in python that finds the best ARIMA model that fits our timeseries data. ARIMA combines autoregression, differencing and moving average.

We chose to scale the data in order to facilitate the differenciation of the timeseries if needed for the ARIMA model

[59]:
```
train_data, test_data = train_test_split(data, test_size=0.045)

sc2 = MinMaxScaler(feature_range=(0, 1))
train_data_scaled = sc2.fit_transform(train_data)
test_data_scaled = sc2.transform(test_data)

model = pm.auto_arima(train_data_scaled, seasonal=True, m=12,␣
 ↪suppress_warnings=True)

print(model.summary())

predictions = model.predict(test_data_scaled.shape[0])
```

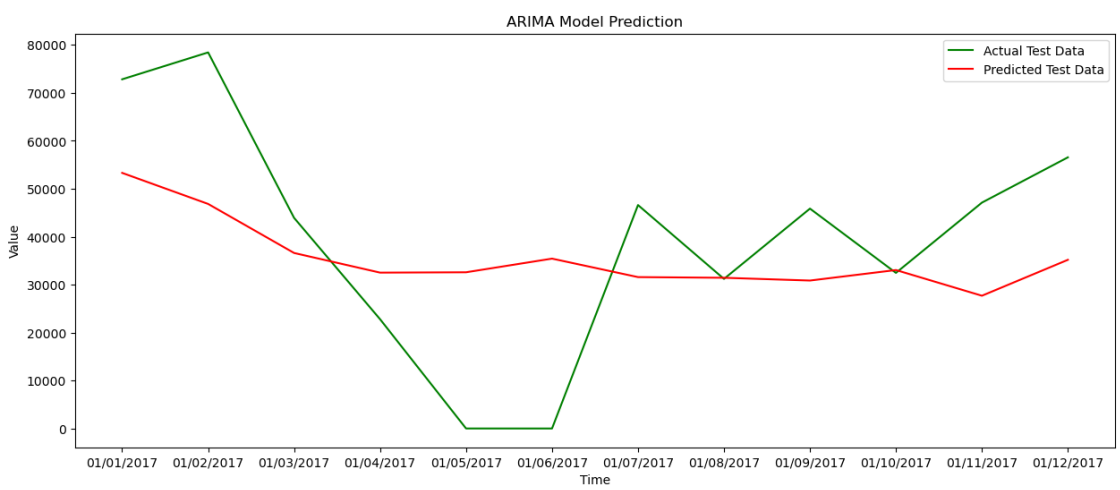
```
                                     SARIMAX Results
==================================================================================
==========
Dep. Variable:                                   y   No. Observations:
252
Model:             SARIMAX(1, 1, 1)x(1, 0, 1, 12)   Log Likelihood
87.930
Date:                            Wed, 29 Nov 2023   AIC
-165.860
Time:                                    15:41:49   BIC
-148.233
Sample:                                         0   HQIC
-158.767
                                            - 252
Covariance Type:                              opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.2172      0.059      3.685      0.000       0.102       0.333
ma.L1         -0.9756      0.022    -43.966      0.000      -1.019      -0.932
ar.S.L12       0.6845      0.113      6.042      0.000       0.462       0.907
ma.S.L12      -0.3920      0.139     -2.814      0.005      -0.665      -0.119
sigma2         0.0286      0.002     16.306      0.000       0.025       0.032
==================================================================================
===
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):
82.67
```

```
Prob(Q):                              1.00   Prob(JB):
0.00
Heteroskedasticity (H):               0.14   Skew:
0.06
Prob(H) (two-sided):                  0.00   Kurtosis:
5.81
================================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
```

p=1: AutoRegressive (AR) terms
d=1: Differencing order (Integrated)
q=1: One Moving Average (MA) terms
P=1: Seasonal AR order (Seasonal AutoRegressive)
D=0: Seasonal differencing order
Q=1: Seasonal MA order
m=12: Seasonal period (monthly data, so 12 for yearly seasonality)

[40]:
```
pred = sc2.inverse_transform(predictions.reshape(-1,1))
```

[41]:
```
plt.figure(figsize=(15, 6))
plt.plot(test_data.index, test_data, label='Actual Test Data', color='green')
plt.plot(test_data.index, pred, label='Predicted Test Data', color='red')
plt.title('ARIMA Model Prediction')
plt.xlabel('Time')
plt.ylabel('Value')
plt.legend()
plt.show()
```



[43]:
```
return_rmse(test_data,pred)
```

```
The root mean squared error is 20666.22.
```

2 Conclusion

On remarque dans notre série qu'à partir de la moitié de l'année 2006, le prix s'est diminué ayant une moyenne plus baisse qu'avant, avec des fluctuations plus réduites.
La série présente une presque saisonnalitér dans chaque année (chaque mois ayant un prix très élevé est suivi par une chute du prix)

Le modèle simpleRNN a montré des performances supérieures par rapport au modèle ARIMA.
La racine carrée de l'erreur quadratique moyenne est plus basse indiquant une meilleure capacité du modèle RNN à comprendre notre série temporelle.

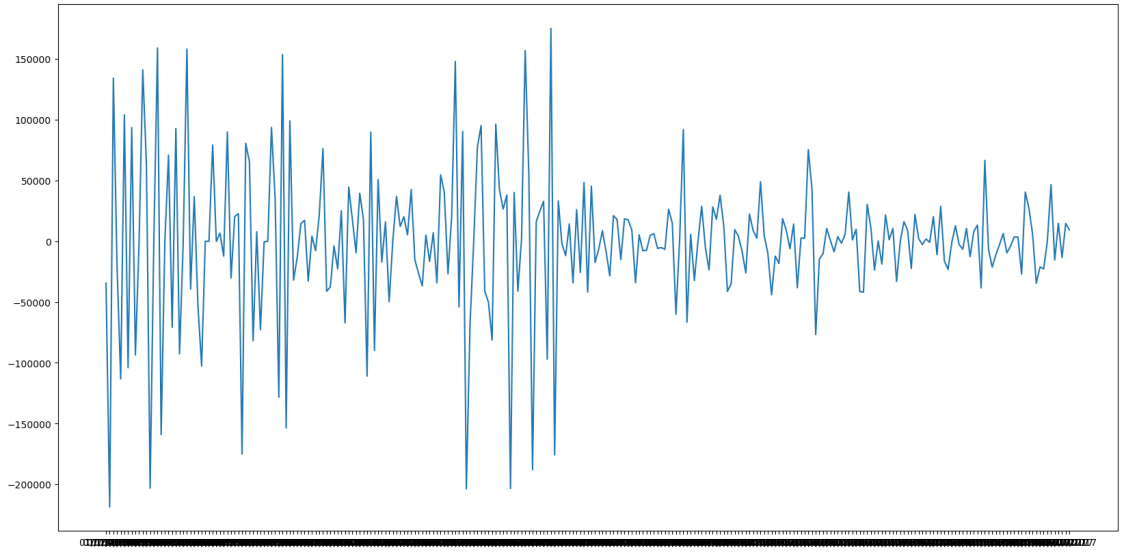
Selon ARIMA model, la serie originale n'est pas stationnaire(ce qui évident visuellement en regardant la graphe au début de ce tp puisuqe les oyennes et variances varient au cours du temps), il faut differencier d'un ordre afin d'atteindre la stationnarité, avec un ordre autoregressive (2 mois consecutifs corrélés) et un ordre MA (erreur).
Ce modèle est déja optimisé avec Python.

Le meilleur modèle RNN obtenu est relativement simple, en utilisant simpleRNN (short memory only), et 2 couches: couche d'entrées de 256 neurones et couche de sortie avec du dropout pour éviter le sur-apprentissage.
Pour que le réseaux de neurones soit capables de capter la saisonnalité à long term, on peut substituer le simpleRNN par LSTM(long and short term memory)

le modèle RNN est plus capable a s'adapter à la compléxité de notre série. Cette dernière n'est pas facilement expliqué par la linéarité avec les observations précédentes (AR) ou erreurs précédents (MA) qui est la base du modèle ARIMA

[30]:
```
differenced_series = data.diff().dropna()
plt.figure(figsize=(20,10))
plt.plot(differenced_series)

from statsmodels.tsa.stattools import adfuller

result = adfuller(differenced_series)
print('ADF Statistic:', result[0])
print('p-value:', result[1])
print('Critical Values:', result[4])
```

```
ADF Statistic: -11.606215893420341
p-value: 2.577341710820297e-21
Critical Values: {'1%': -3.4565688966099373, '5%': -2.8730786194395455, '10%':
-2.5729189953388762}
```



On voit que meme après la differenciation d'ordre 1 suggéré par la fonction auto_arima, la série présente des fluctuations (std) qui diffèrent au cours du temps.

```
[ ]:
```