7_Question_2

November 9, 2023

1 EE4211 Project Group 7

2 Outline

Part 2.1: Written answer for Q2.1
Pre 2.2: Obtaining dataset from July and Aug (CSV included in submission file, not necessary to run this cell again)
Pre 2.2: Processing the raw data to obtain the timestamp and availability percentage for July and Aug (Start running from this cell)
Part 2.2: Linear Regression model and plots (included sliding window method)
Part 2.3: Support Vector Regressor model and plots
Part 2.4: Decision Tree Regressor model and plots
Part 2.5: Recommendation of best regression model based on MSE

2.0.1 Part 2.1

In this part, you will build a model to forecast the hourly carpark availability in the future (averaged across all carparks instead of looking at each carpark individually). Can you explain why you may want to forecast the carpark availability in the future? Who would find this information valuable? What can you do if you have a good forecasting model?

Forcasting carpark availability enables traffic optimization and the development of carpark infrastructure, which is valuable for the government and urban infrastructure planners. By estimating the number of cars in specific regions, stakeholders gains awareness of the regions with higher vehicle density compared to the rest. This insight enables them to prioritize these regions for road alterations, widening road to facilitate smoother traffic, and and building additional carparks to reduce stress and frustation associated with being unable to find a parking spot.

A reliable forecasting model, characterized by highly accurate and credible data, ensures that the resources are being utilised properly. Investing in areas with lower vehicle density might not yield proportional returns, highlighting the importance of effective resource allocation to areas who would benefit more significantly than others.

2.0.2 Necessary imports in this ipynb file

```
[ ]: import requests
     import json
     import pandas as pd
     import numpy as np
```

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
```

2.0.3 Pre 2.2: Extracting necessary dataset

```
[ ]: # Reusing function from Part 1 to obtain data
def get_data(year, month, day, hour, minute, second):
    site = f'https://api.data.gov.sg/v1/transport/carpark-availability?
 ↪date_time={year}-{month.zfill(2)}-{day.zfill(2)}T{hour.zfill(2)}%3A{minute.
 ↪zfill(2)}%3A{second.zfill(2)}'
    response_API = requests.get(site)
    parsed_data = json.loads(response_API.text)
    df = pd.DataFrame()

    if "items" in parsed_data and parsed_data["items"]:
        data = parsed_data["items"][0]["carpark_data"]
        df = pd.DataFrame(data)
        for heading in ("total_lots", "lot_type", "lots_available"):
            df[heading] = df["carpark_info"].apply(lambda x: x[0][heading])
        df = df.drop(["carpark_info"], axis=1)
    return df

def data_of_month(month):
    data_records = []
    for day in range(1, 32):  # Assuming 31 days for each month, adjust if␣
 ↪necessary
        print(f"Checking data for 2022-{month}-{day:02d}")  # Progress check
        for hour in range(24):
            hourly_df = get_data("2022", month, str(day), str(hour), "00", "00")

            datetime_str = f"2022-{month}-{day:02d} {hour:02d}:00:00"

            if hourly_df.empty:
                data_records.append((datetime_str, None))
            else:
                # Convert columns to numeric and compute availability percentage
                hourly_df['lots_available'] = pd.
 ↪to_numeric(hourly_df['lots_available'], errors='coerce')
                hourly_df['total_lots'] = pd.
 ↪to_numeric(hourly_df['total_lots'], errors='coerce')
                hourly_df['availability_percentage'] =␣
 ↪hourly_df['lots_available'] / hourly_df['total_lots'] * 100
                avg_availability = hourly_df['availability_percentage'].mean()
                data_records.append((datetime_str, avg_availability))

    # Convert the list of tuples into a DataFrame
    df = pd.DataFrame(data_records, columns=['datetime',␣
↪'availability_percentage'])
    return df
```

[ ]:
```
# For July
July_df = data_of_month("07")
July_df.to_csv('July.csv', index=False)
print(July_df)
```

```
Checking data for 2022-07-01
Checking data for 2022-07-02
Checking data for 2022-07-03
Checking data for 2022-07-04
Checking data for 2022-07-05
Checking data for 2022-07-06
Checking data for 2022-07-07
Checking data for 2022-07-08
Checking data for 2022-07-09
Checking data for 2022-07-10
Checking data for 2022-07-11
Checking data for 2022-07-12
Checking data for 2022-07-13
Checking data for 2022-07-14
Checking data for 2022-07-15
Checking data for 2022-07-16
Checking data for 2022-07-17
Checking data for 2022-07-18
Checking data for 2022-07-19
Checking data for 2022-07-20
Checking data for 2022-07-21
Checking data for 2022-07-22
Checking data for 2022-07-23
Checking data for 2022-07-24
Checking data for 2022-07-25
Checking data for 2022-07-26
Checking data for 2022-07-27
Checking data for 2022-07-28
Checking data for 2022-07-29
Checking data for 2022-07-30
Checking data for 2022-07-31
                datetime  availability_percentage
0    2022-07-01 00:00:00                      NaN
1    2022-07-01 01:00:00                43.294615
2    2022-07-01 02:00:00                42.631265
```

```
3    2022-07-01 03:00:00                42.317495
4    2022-07-01 04:00:00                42.386347
..                   …                        …
739  2022-07-31 19:00:00                48.133001
740  2022-07-31 20:00:00                47.472854
741  2022-07-31 21:00:00                46.708942
742  2022-07-31 22:00:00                45.784997
743  2022-07-31 23:00:00                44.784644

[744 rows x 2 columns]
```

[ ]:
```
# For August
Aug_df = data_of_month("08")
Aug_df.to_csv('Aug.csv', index=False)
print(Aug_df)
```

```
Checking data for 2022-08-01
Checking data for 2022-08-02
Checking data for 2022-08-03
Checking data for 2022-08-04
Checking data for 2022-08-05
Checking data for 2022-08-06
Checking data for 2022-08-07
Checking data for 2022-08-08
Checking data for 2022-08-09
Checking data for 2022-08-10
Checking data for 2022-08-11
Checking data for 2022-08-12
Checking data for 2022-08-13
Checking data for 2022-08-14
Checking data for 2022-08-15
Checking data for 2022-08-16
Checking data for 2022-08-17
Checking data for 2022-08-18
Checking data for 2022-08-19
Checking data for 2022-08-20
Checking data for 2022-08-21
Checking data for 2022-08-22
Checking data for 2022-08-23
Checking data for 2022-08-24
Checking data for 2022-08-25
Checking data for 2022-08-26
Checking data for 2022-08-27
Checking data for 2022-08-28
Checking data for 2022-08-29
Checking data for 2022-08-30
Checking data for 2022-08-31
                datetime  availability_percentage
```

```
0    2022-08-01 00:00:00                      NaN
1    2022-08-01 01:00:00                43.494269
2    2022-08-01 02:00:00                43.076411
3    2022-08-01 03:00:00                42.918657
4    2022-08-01 04:00:00                42.617799
..                   …                        …
739  2022-08-31 19:00:00                53.168277
740  2022-08-31 20:00:00                49.364047
741  2022-08-31 21:00:00                47.334533
742  2022-08-31 22:00:00                46.363437
743  2022-08-31 23:00:00                44.709454

[744 rows x 2 columns]
```

2.0.4 Pre 2.2: Process the raw data

[ ]:

```
# Interpolate and save to a dataframe
def interpolate_data(filename):
  df = pd.read_csv(filename)
  interpolated_column = df.iloc[:, 1].interpolate(limit_direction='both')
  df.iloc[:,1] = interpolated_column
  return df

interpolated_july = interpolate_data("July.csv")
interpolated_aug = interpolate_data("Aug.csv")

data = {
    'datetime': pd.date_range(start="2022-07-01",␣
 ↪periods=len(interpolated_july), freq="H"),
    'availability_percentage': interpolated_july['availability_percentage']
}
July_df = pd.DataFrame(data)

data = {
    'datetime': pd.date_range(start="2022-08-01",␣
 ↪periods=len(interpolated_aug), freq="H"),
    'availability_percentage': interpolated_aug['availability_percentage']
}
Aug_df = pd.DataFrame(data)

X_July = np.arange(len(July_df)).reshape(-1, 1)  # Use integer values as an␣
 ↪alternative to datetime
y_July = July_df['availability_percentage'].values

X_Aug = np.arange(len(Aug_df)).reshape(-1, 1)  # Use integer values as an␣
 ↪alternative to datetime
y_Aug = Aug_df['availability_percentage'].values
```

2.0.5 Part 2.2 (LR)

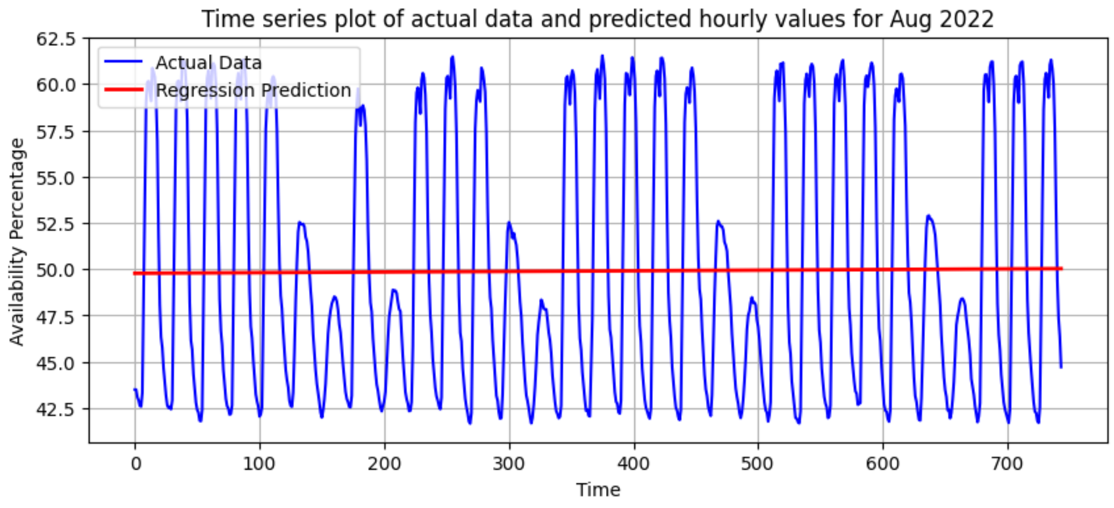
Linear Regression Plot Build a linear regression model to forecast the hourly carpark availability for a given month. Use the month of July 2022 as a training dataset and the month of August 2022 as the test dataset. For this part, do not use additional datasets. The target is the hourly carpark availability percentage and you will have to decide what features you want to use.

Generate two plots:
(i) Time series plot of the actual and predicted hourly values
(ii) Scatter plot of actual vs predicted hourly values (along with a line showing how good the fit is).

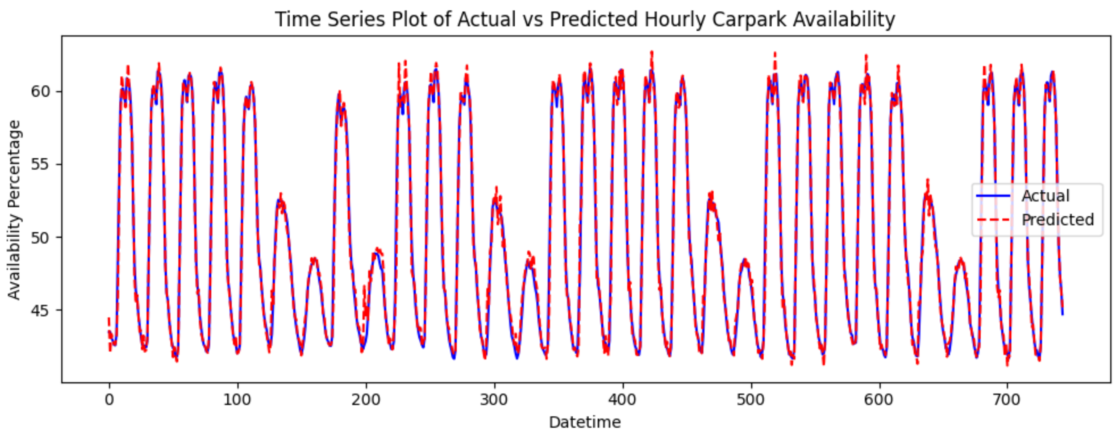
```
[ ]: # Create and train the Linear Regression model
lr = LinearRegression()
lr.fit(X_July, y_July)
y_pred = lr.predict(X_Aug)

# Calculate and print the mean squared error (MSE)
mse = mean_squared_error(y_Aug, y_pred)
print("Mean squared error (LR):", mse)

# Plot the prediction
plt.figure(figsize=(10,4))
plt.plot(X_Aug, y_Aug, color='blue', label='Actual Data')
plt.plot(X_Aug, y_pred, color='red', linewidth=2, label='Regression Prediction')
plt.xlabel('Time')
plt.ylabel('Availability Percentage')
plt.title("Time series plot of actual data and predicted hourly values for Aug␣
  ↪2022")
plt.legend(loc='upper left')
plt.grid(True)
plt.show()

########## SLIDING WINDOW ##########
'''
Data:
X_July: Integer values as an alternative to datetime
y_July: Availability percentage for July
X_Aug: Integer values as an alternative to datetime
y_Aug: Availability percentage for Aug

In this method:
X: Historical n hours of availability percentage, where n = window size
y: Current availability percentage
'''

# avail_percentage = y_July.tolist()
# window_sizes = [10,24,50,126,300]
# mse_values = []

# for window_size in window_sizes:
#     X = []
#     y = []

#     for i in range(len(avail_percentage)-window_size):
#         X.append(avail_percentage[i:i+window_size])
#         y.append(avail_percentage[i+window_size])

#     # create a Linear Regression model
#     lr = LinearRegression()
#     lr.fit(X, y)

#     X2 = []
#     for i in range(len(avail_percentage)-window_size):
#         X2.append(avail_percentage[i:i+window_size])
#     y_pred = lr.predict(X2)

#     mse = mean_squared_error(y_July[window_size:],y_pred)
#     mse_values.append(mse)

#     # Visualize the original and predicted data
#     actual_avail_percentage = y_July
#     predicted_avail_percentage = y_pred
#     plt.figure(figsize=(10, 3))
#     plt.plot(X_July, actual_avail_percentage, label='Actual', color='blue')
#     plt.plot(X_July[window_size:], predicted_avail_percentage,␣
 ↪label='Predicted', color='red')
#     plt.xlabel('Time')
#     plt.ylabel('Avail_percentage')
#     plt.title("Time series plot of actual data and predicted hourly values␣
 ↪for July 2022")
#     plt.legend(loc='upper left')
#     plt.grid(True)
#     plt.show()

# for i in range(len(window_sizes)):
#     print("Window size: {:^5}, MSE value: {:^10}".
 ↪format(window_sizes[i],mse_values[i]) )
# plt.scatter(window_sizes,mse_values)
# plt.plot(window_sizes,mse_values)
# plt.grid(True)
# plt.show()

'''
```

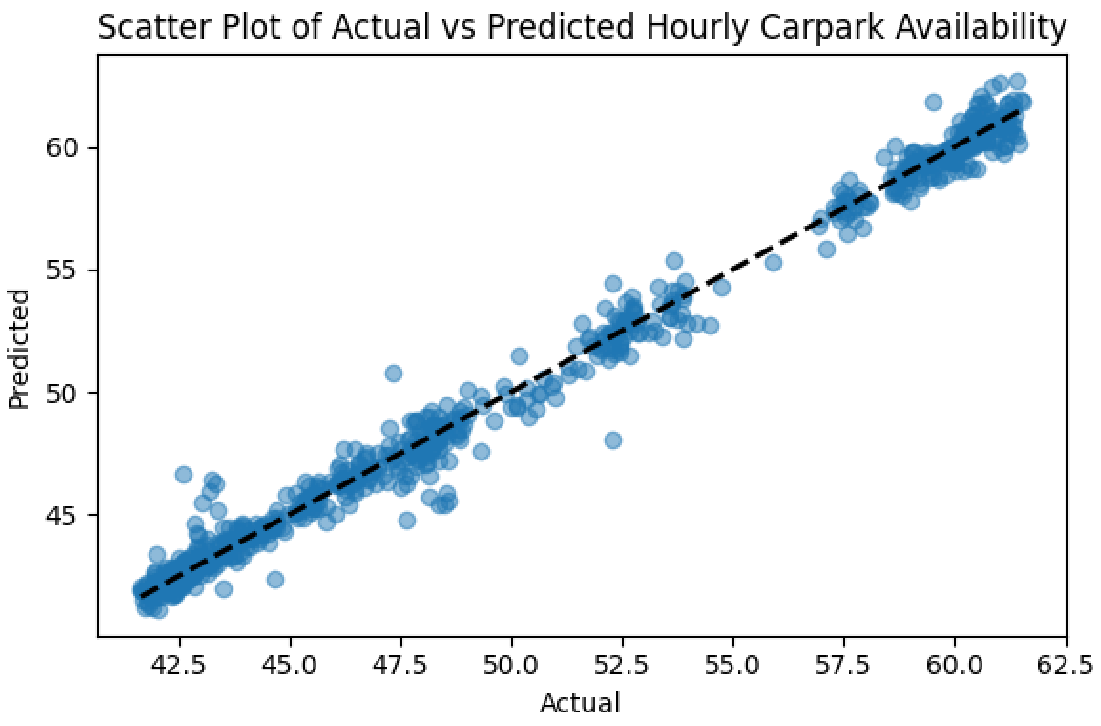
```
Selected Window Size of 126 ~ 1 weeks worth of data
MSE improvements slows down beyond 126, even though it still improves,
we did not want to overfit or have the possibility of loss of relevance
'''

# Chosen window size = 126
window_size = 126

X = []
y = []

avail_percentage = y_July.tolist()

for i in range(len(avail_percentage)-window_size):
    X.append(avail_percentage[i:i+window_size])
    y.append(avail_percentage[i+window_size])

# create a Linear Regression model and train using sliding window method
lr = LinearRegression()
lr.fit(X, y)

dataset = y_July[-window_size:].tolist() + y_Aug.tolist()
X2 = []
for i in range(len(dataset)-window_size):
    X2.append(dataset[i:i+window_size])

LRSW_predictions = lr.predict(X2)

LRmse_window = mean_squared_error(y_Aug,LRSW_predictions)
print("\nWith a window size of {}, the MSE value for linear regression using␣
 ↪windowing method is: {}".format(window_size,LRmse_window))

# Plotting
# Time series plot of the actual and predicted hourly values
plt.figure(figsize=(10, 4))
plt.plot(X_Aug, y_Aug, label="Actual", color="blue")
plt.plot(X_Aug, LRSW_predictions, label="Predicted", color="red",␣
 ↪linestyle="--")
plt.xlabel("Datetime")
plt.ylabel("Availability Percentage")
plt.title("Time Series Plot of Actual vs Predicted Hourly Carpark Availability")
plt.legend()
plt.tight_layout()
plt.show()

# Scatter plot of actual vs predicted hourly values
plt.figure(figsize=(6,4))
plt.scatter(y_Aug, LRSW_predictions, alpha=0.5)
plt.plot([min(y_Aug), max(y_Aug)], [min(y_Aug), max(y_Aug)], 'k--', lw=2)
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.title("Scatter Plot of Actual vs Predicted Hourly Carpark Availability")
plt.tight_layout()
plt.show()
```

```
Mean squared error (LR): 48.232793929558824
```



```
With a window size of 126, the MSE value for linear regression using windowing
method is: 0.4859490159654106
```





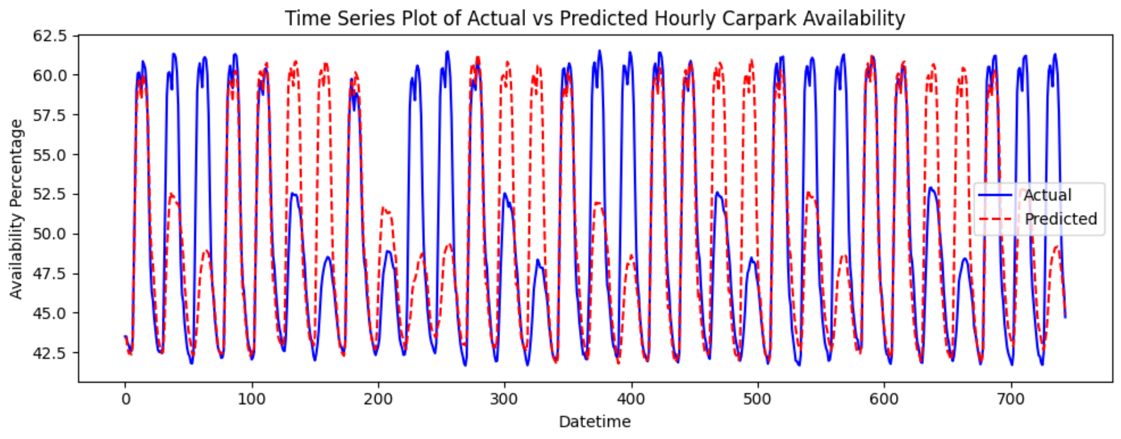
2.0.6 Part 2.3 (SVR)

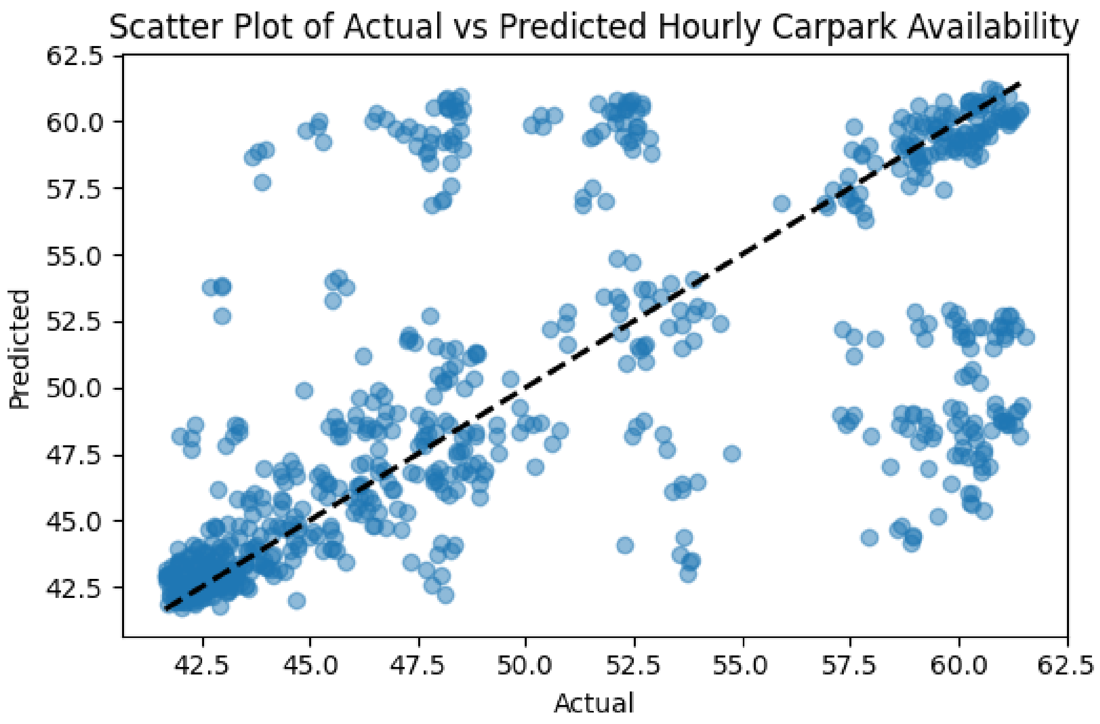
Build a support vector regressor (SVR) model to forecast the hourly carpark availability for a given month. Use the month of July 2022 as a training dataset and the month of August 2022 as the test dataset. For this part, do not use additional datasets. The target is the hourly carpark availability percentage and you will have to decide what features you want to use.

Generate two plots:
(i) Time series plot of the actual and predicted hourly values
(ii) Scatter plot of actual vs predicted hourly values (along with a line showing how good the fit is).

[ ]:
```
# Train:0.8 Test Split:0.2 on July Data
X_train, X_test, y_train, y_test = train_test_split(X_July, y_July, test_size=0.
 ↪2, random_state=5)

temp_data = {
  'datetime': X_test.tolist(),
  'availability_percentage': y_test.tolist()
}
temp_df = pd.DataFrame(temp_data)
sorted_df = temp_df.sort_values(by='datetime')
X_test = sorted_df['datetime'].tolist()
y_test = sorted_df['availability_percentage'].tolist()
```

```
temp_data = {
    'datetime': X_train.tolist(),
    'availability_percentage': y_train.tolist()
}
temp_df = pd.DataFrame(temp_data)
sorted_df = temp_df.sort_values(by='datetime')
X_train = sorted_df['datetime'].tolist()
y_train = sorted_df['availability_percentage'].tolist()

# Hypertuning based on July Data Set
# for i in [1,5,10,50,100]:
#   model = SVR(C=i, kernel='rbf',gamma='auto')
#   model.fit(X_train, y_train)
#   predictions = model.predict(X_test)

#   # Calculate and print the mean squared error (MSE)
#   mse = mean_squared_error(y_test, predictions)
#   print('MSE with C value, ', i)
#   print("Mean squared error:", mse)

#   plt.figure(figsize=(7, 3))
#   plt.plot(X_July, y_July, label="Actual", color="blue")
#   plt.plot(X_test, predictions, label="Predicted", color="red",␣
 ↪linestyle="--")
#   plt.xlabel("Datetime")
#   plt.ylabel("Availability Percentage")
#   plt.title("Time Series Plot of Actual vs Predicted Hourly Carpark␣
 ↪Availability")
#   plt.legend()
#   plt.tight_layout()
#   plt.show()

'''
Hypertuning Done -> Select parameter C = 10 because MSE is 12.493989901124813
C = 10 sees a much bigger improvement as compared to C = 5 in terms of MSE,
But very little improvements can be see from C = 10 onwards
'''

model = SVR(C=10, kernel='rbf',gamma='auto')
model.fit(X_July, y_July)
SVRpredictions = model.predict(X_Aug)

# Plotting
# Time series plot of the actual and predicted hourly values
plt.figure(figsize=(10, 4))
plt.plot(X_Aug, y_Aug, label="Actual", color="blue")
plt.plot(X_Aug, SVRpredictions, label="Predicted", color="red", linestyle="--")
plt.xlabel("Datetime")
plt.ylabel("Availability Percentage")
plt.title("Time Series Plot of Actual vs Predicted Hourly Carpark Availability")
plt.legend()
plt.tight_layout()
plt.show()

# Scatter plot of actual vs predicted hourly values
plt.figure(figsize=(6, 4))
plt.scatter(y_Aug, SVRpredictions, alpha=0.5)
plt.plot([min(y_Aug), max(y_Aug)], [min(y_Aug), max(y_Aug)], 'k--', lw=2)
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.title("Scatter Plot of Actual vs Predicted Hourly Carpark Availability")
plt.tight_layout()
plt.show()
```





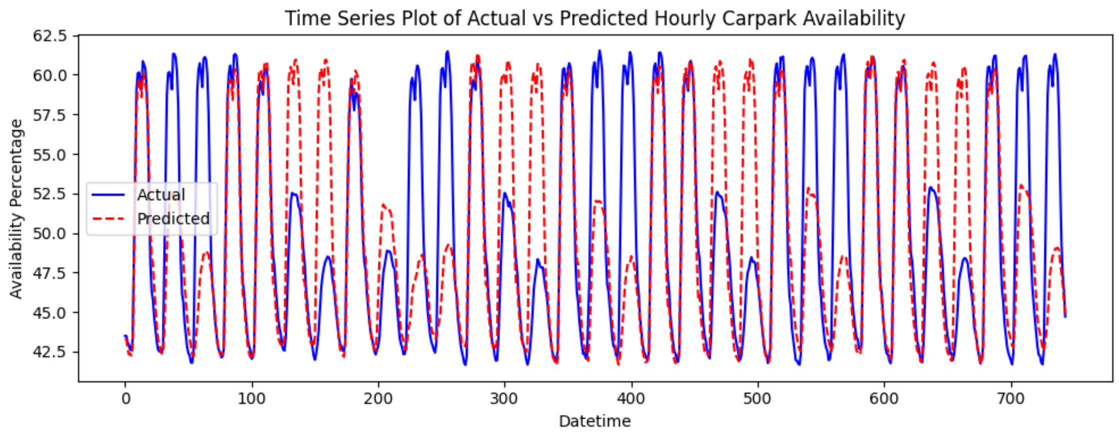
2.0.7 Part 2.4 (DT)

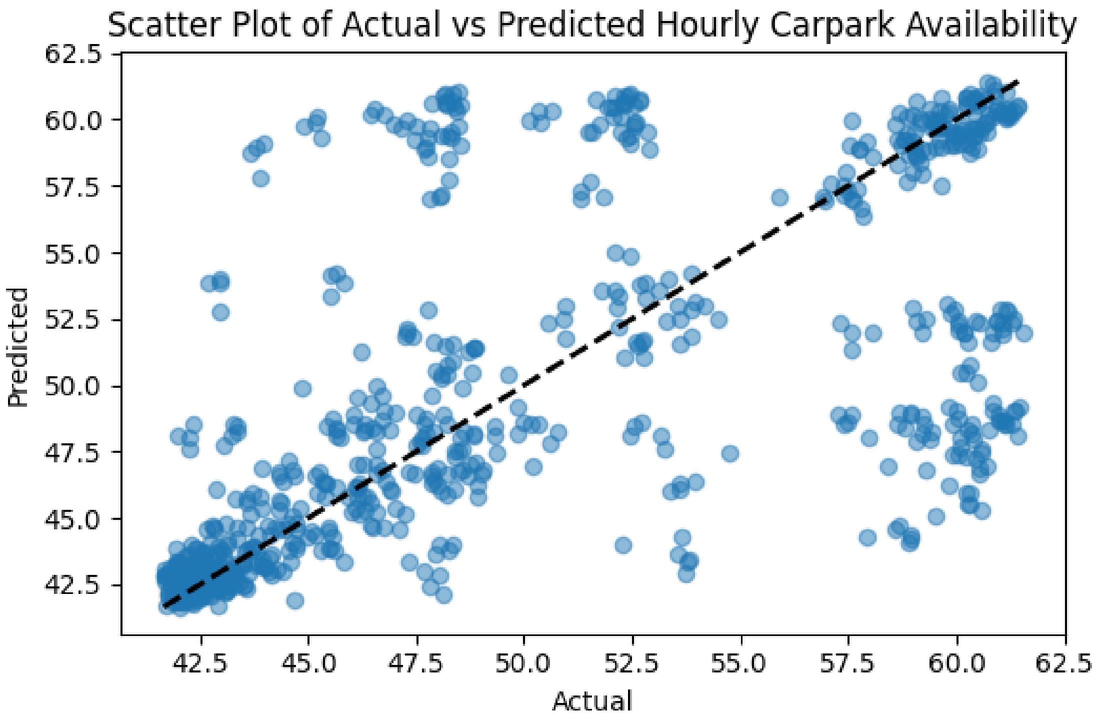
Build a decision tree (DT) regressor model to forecast the hourly carpark availability for a given month. Use the month of July 2022 as a training dataset and the month of August 2022 as the test dataset. For this part, do not use additional datasets. The target is the hourly carpark availability percentage and you will have to decide what features you want to use.

Generate two plots:
(i) Time series plot of the actual and predicted hourly values
(ii) Scatter plot of actual vs predicted hourly values (along with a line showing how good the fit is).

[ ]:
```
# Train:0.8 Test Split:0.2 on July Data
X_train, X_test, y_train, y_test = train_test_split(X_July, y_July, test_size=0.
↪2, random_state=5)

temp_data = {
  'datetime': X_test.tolist(),
  'availability_percentage': y_test.tolist()
}
temp_df = pd.DataFrame(temp_data)
sorted_df = temp_df.sort_values(by='datetime')
X_test = sorted_df['datetime'].tolist()
y_test = sorted_df['availability_percentage'].tolist()
```

```
temp_data = {
    'datetime': X_train.tolist(),
    'availability_percentage': y_train.tolist()
}
temp_df = pd.DataFrame(temp_data)
sorted_df = temp_df.sort_values(by='datetime')
X_train = sorted_df['datetime'].tolist()
y_train = sorted_df['availability_percentage'].tolist()

# for i in [5, 10, 20, 50, 60, 80, 100]:
#     DTmodel = DecisionTreeRegressor(max_depth = i)
#     DTmodel.fit(X_train, y_train)
#     DTpredictions = DTmodel.predict(X_test)

#     # Calculate and print the mean squared error (MSE)
#     mse = mean_squared_error(y_test, DTpredictions)
#     print('MSE with DT value, ', i)
#     print("Mean squared error:", mse)

#     plt.figure(figsize=(7, 3))
#     plt.plot(X_July, y_July, label="Actual", color="blue")
#     plt.plot(X_test, DTpredictions, label="Predicted", color="red",␣
 ↪linestyle="--")
#     plt.xlabel("Datetime")
#     plt.ylabel("Availability Percentage")
#     plt.title("Time Series Plot of Actual vs Predicted Hourly Carpark␣
 ↪Availability")
#     plt.legend()
#     plt.tight_layout()
#     plt.show()

'''
Hypertuning Done -> Select parameter DT = 50 because MSE is 4.7659050365666005
DT = 50 sees a much bigger improvement as compared to DT = 20 in terms of MSE,
But very little improvements can be see from DT = 50 onwards
'''

DTmodel = DecisionTreeRegressor(max_depth = 50)
DTmodel.fit(X_July, y_July)
DTpredictions = DTmodel.predict(X_Aug)

# Plotting
# Time series plot of the actual and predicted hourly values
plt.figure(figsize=(10, 4))
plt.plot(X_Aug, y_Aug, label="Actual", color="blue")
plt.plot(X_Aug, DTpredictions, label="Predicted", color="red", linestyle="--")
plt.xlabel("Datetime")
plt.ylabel("Availability Percentage")
plt.title("Time Series Plot of Actual vs Predicted Hourly Carpark Availability")
plt.legend()
plt.tight_layout()
plt.show()

# Scatter plot of actual vs predicted hourly values
plt.figure(figsize=(6, 4))
plt.scatter(y_Aug, DTpredictions, alpha=0.5)
plt.plot([min(y_Aug), max(y_Aug)], [min(y_Aug), max(y_Aug)], 'k--', lw=2)
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.title("Scatter Plot of Actual vs Predicted Hourly Carpark Availability")
plt.tight_layout()
plt.show()
```





2.0.8 Part 2.5

Make a final recommendation for the best regression model (out of the 3 methods above) by choosing a suitable performance metric. To ensure a fair comparison, carry out hyper- parameter tuning for all 3 methods. Then, make a final recommendation selecting only one model. Include both quantitative and qualitative arguments for your choice.

[ ]:
```
# MSE Analysis Between ACTUAL AUGUST AND PREDICTED AUGUST
# Calculate and print the mean squared error (MSE)
# LR MSE
LRmse = mean_squared_error(y_Aug, y_pred)
print('LR MSE performance between Actual and Prediction is ', LRmse)

# LR MSE - Sliding Window
LRmse_window = mean_squared_error(y_Aug, LRSW_predictions)
print('LR Sliding Window MSE performance between Actual and Prediction is ',␣
  ↪LRmse_window)

# SVR MSE
SVRmse = mean_squared_error(y_Aug, SVRpredictions)
print('SVR MSE performance between Actual and Prediction is ', SVRmse)

#DT MSE
```

```python
DTmse = mean_squared_error(y_Aug, DTpredictions)
print('DT MSE performance between Actual and Prediction is ', DTmse)
```

```
LR MSE performance between Actual and Prediction is  48.232793929558824
LR Sliding Window MSE performance between Actual and Prediction is
0.4859490159654106
SVR MSE performance between Actual and Prediction is  32.10378905056394
DT MSE performance between Actual and Prediction is  32.5097424015183
```

Our team's approach to determining the best regression model was done by using MSE as a metric. This is because it measures the average squared difference between predicted and actual values and penalizes larger error more heavily. This metric is well suited to what we have been doing in Q2 where we plot and compare our Prediction and Actual results.

The procedures that were carried in Q2 to determine the best regression model was: 1. Create the models for LR, SVR, DT by training and testing using train-test split of July data - initially default parameters 2. In order to carry out hyper-parameter tuning, we ran For-loops with varying hyper-parameter values for LR (sliding window size), SVR (C parameter), DT (max depth), and got the MSE of the predicted value against the July test set 3. We then selected the best model hyper-paramter, and trained the model with the whole July dataset with the selected parameter, and then tested with August dataset. 4. Here in Q2.5 we then checked the MSE of the August Actual values against the Predicted values

Based on the results Obtained:
LR MSE performance between Actual and Prediction is 48.232793929558824
LRSW MSE performance between Actual and Prediction is 0.48594901596540985
SVR MSE performance between Actual and Prediction is 32.10378905056394
DT MSE performance between Actual and Prediction is 32.5097424015183

Based on these results, using a the Linear regression with sliding window, yields a better result as the MSE is: 0.48594901596540985.

The reason why applying sliding window to the problem we are trying to solve is effective is due to the fact that the data we are dealing with is a time-series dataset. In datasets with time-related structure or order, the sequence or chronological order of observations matters, utilising the sliding window method allows our model to adapt to changes in the data distribution as it moves through different time periods.

Sliding window is not limited to be used on Linear Regression, SVR could also utilise such methodology, it might not be as applicable. Although we only incorporated it to our Linear Regression model.