CS7CS4 - Machine Learning Final Assignment Report

Date: 05/01/2023 **Name:** Yuanpei Teng **Student Number:** 17343768

PART 1.

A. Data Preprocessing and Feature Engineering

This project contains the datasets related to Dublin bike usage data from 2018 to 2023. However, each year's data is collected and organized differently. After looking at the datasets for each year, here is the first step before dive into detail data preprocessing, which is gain a general overview of the structure of these datasets:

	Year	Rows	Columns
0	2018	4937925	11
1	2019	10741877	11
2	2020	11023075	11
3	2021	11283524	11
4	2022	1950289	11
5	2023	1983684	11

2018: Datasets are collected on a quarterly basis, only Q3 and Q4 were recorded. Each row of data is recorded at **5 minutes** granularity.

2019 to 2021: 3 years datasets are collected on a quarterly basis. Each row of data is recorded at **5 minutes** granularity.

2022 and 2023: These two datasets are collected monthly and recorded at **30 minutes** granularity.

Following are the steps done to preprocess these data frames and perform feature engineering:

- **Uniform name of columns:**
  Firstly, I found three columns of names from 2018 to 2021 that do not contain underscores, they are: 'BIKE STANDS', 'AVAILABLE BIKE STANDS' and 'AVAILABLE BIKES', but in the data for 2022 to 2023, these columns do contain underscores, so for the sake of uniformity in the dataset formatting, I uniformly added underscores to the three column names mentioned above, which became 'BIKE_STANDS', 'AVAILABLE_BIKE_STANDS' and 'AVAILABLE_BIKES'.
- **Converting data formats and extract new time features:**
  Convert time-related columns like 'TIME' and 'LAST UPDATED' to datetime format. By doing that we can extract time units in any format. Therefore, 6 new columns are extracted from 'TIME' column, they are 'DATE', 'WEEKDAY', 'YEAR', 'MONTH', 'DAY' and 'HOUR'.
- **Checking NaN values and drop if exists.**
  There are no NaN values in these data frames. Here below is the statistic report for collecting total amount of missing values for each column, all zeros mean no missing values.

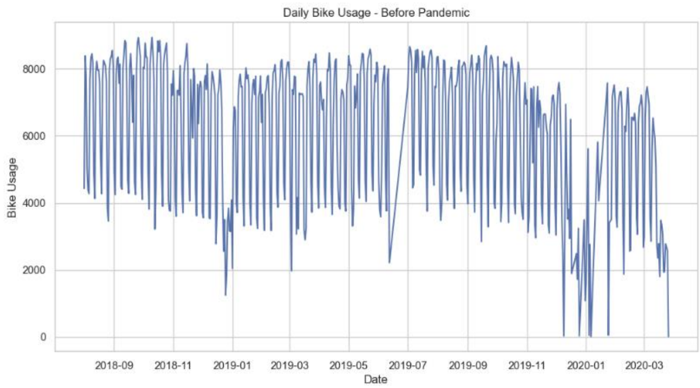
	2018	2019	2020	2021	2022	2023
STATION ID	0	0	0	0	0	0
TIME	0	0	0	0	0	0
LAST UPDATED	0	0	0	0	0	0
NAME	0	0	0	0	0	0
BIKE_STANDS	0	0	0	0	0	0
AVAILABLE_BIKE_STANDS	0	0	0	0	0	0
AVAILABLE_BIKES	0	0	0	0	0	0
STATUS	0	0	0	0	0	0
ADDRESS	0	0	0	0	0	0
LATITUDE	0	0	0	0	0	0
LONGITUDE	0	0	0	0	0	0
DATE	0	0	0	0	0	0
WEEKDAY	0	0	0	0	0	0
YEAR	0	0	0	0	0	0
MONTH	0	0	0	0	0	0
DAY	0	0	0	0	0	0
HOUR	0	0	0	0	0	0

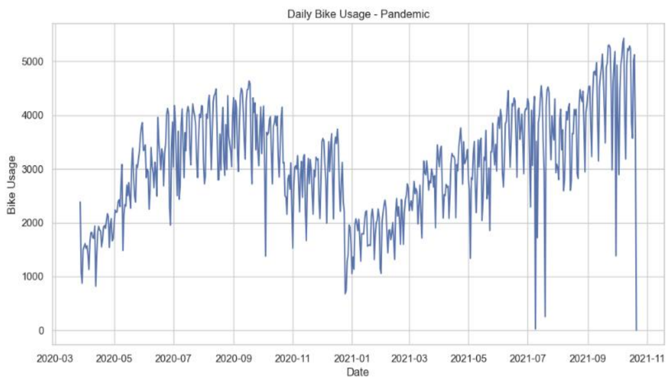
- **Combine different data frames based on Pandemic timeline.**
  Since our aim is to assess the impact of Pandemic on bike usages, therefore, our datasets aggregated into three

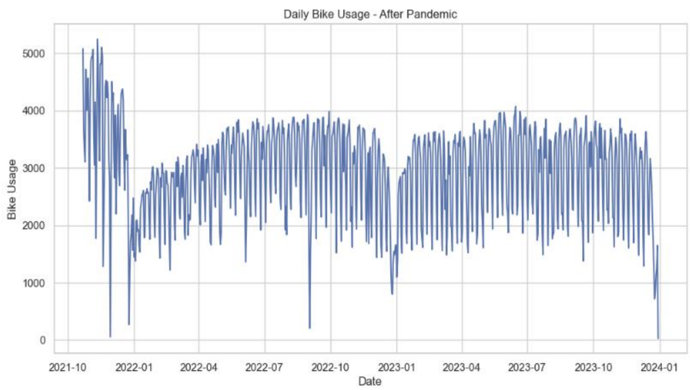
stages: **Before Pandemic (2018-01-01 to 2020-03-26)**, **Pandemic (2020-03-27 to 2021-10-22)**, **After Pandemic (2021-10-23 to 2023-12-30)**. Each stage is a separate data frame, and we also have one data frame that contains all the data from all other data frames.

- **Mining new features based on the features provided by the current dataset.**
  Since our task is to study the impact of the pandemic on bike usage, bike usage is the column we want to focus on, but this dataset doesn't give the relevant quantities directly, but by subtracting the 'AVAILABLE_BIKES' data from the next row using the 'AVAILABLE_BIKES' of the previous row, the difference is the number of bike used during this period of time, and if the result is a negative number, it means that the bikes are being returned rather than being used, so we will set any negative result as 0 in the newly created 'BIKE_USAGE' column.
- **Drop insignificant features.**
  In the current dataset, time is strongly correlated with our predictive features, so we chose to keep the time-correlated features and delete the features that are less helpful for prediction.
  Dropped columns: 'LAST_UPDATED', 'BIKE_STANDS', 'AVAILABLE_BIKE_STANDS', 'AVAILABLE_BIKES', 'STATUS', 'ADDRESS','LATITUE','LONGITUE'.
- **Data aggregation**
  Because this prediction task is yearly level, and each dataset is currently at the minute level, the data is too large for later model training, so I aggregated all the datasets on an hourly basis to change the granularity of the data from the minute level to the hourly level and calculated the amount of the bike used for each hour of the day.
- **Retrieve general trends on Data.**
  After finishing the data preprocessing on the origin data, it is necessary to get a detail statistical report relate to the data, that may provide user a good insights on data. Therefore, I calculated statistics related to 'BIKE_USAGE' by three periods of time, **before-pandemic**, **pandemic**, and **after-epidemic**, and drew separate line graphs of the number of bikes used per day to facilitate a better understanding of the overall trend of the data.

	Before_pandemic	Pandemic	After_pandemic
Total_usage	3470196.00	1777381.00	2341480.00
Average_hourly_usage	263.25	129.90	123.13
Max_daily_usage	8936.00	5429.00	5248.00
Min_daily_usage	12.00	0.00	27.00
Average_daily_usage	5745.36	3096.48	2926.85
Average_weekly_usage	495742.29	253911.57	334497.14
Average_monthly_usage	289183.00	148115.08	195123.33
Median	233.00	126.00	117.00
Standard Deviation	224.70	101.23	97.41







- **Insert weather data to the dataset.**
  Since the usage of bikes is correlated with weather conditions, adding relevant weather information may be helpful for the model's prediction accuracy. Therefore, I added a new feature **'precip'** to the dataframe, it indicates the rainfall level in Dublin hourly. And since the 'precip' is numeric, to better use this feature in model, I categorized the values into five categorical data based on the amount of rainfall, which are 'No Rain', 'Light Rain', 'Moderate Rain', 'Heavy Rain' and 'Violate Rain'. I stored these categorial data into a new column called **'precip_level'**.

- **Label Encoding**

  I used Label Encoder from sklearn.preprocessing to encode the categorical feature ‘precip_level’. Since ‘precip_level’ contains 5 elements, these values are converted to ‘0’, ‘1’, ‘2’, ‘3’, ‘4’ and stored into a new column **'label_encoded_preciplevel',** with the help of encoding, the precipitation features can be included in the model.

- **Training Features Selection**

  After merge the data from the weather dataset, the dataframe used for model training now contains 10 columns, as example shown below:

	YEAR	MONTH	DAY	HOUR	WEEKDAY	BIKE_USAGE	TIME	precip	precip_level	label_encoded_preciplevel
0	2018	8	1	12	2	94	2018-08-01 12:00:00	0.000	No Rain	3
1	2018	8	1	13	2	261	2018-08-01 13:00:00	2.528	Moderate Rain	2

To select features, I applied **random forest** model as a test model, to test different combination of features and recorded the R^2 score of each model, higher R^2 score indicates higher model predict accuracy. Since this project is a time-series prediction task, any features relate to time is highly important, so the three different feature combinations are:

1. ['YEAR', 'MONTH', 'DAY', 'HOUR'] **R^2 score:** 0.7078
2. ['YEAR', 'MONTH', 'DAY', 'WEEKDAY', 'HOUR'] **R^2 score:** 0.9448
3. ['YEAR', 'MONTH', 'DAY', 'WEEKDAY', 'HOUR','label_encoded_preciplevel'] **R^2 score:** 0.9446.

Based on the R^2 score, we can see that ‘weekday’ is a very important features, since it improved the R^2 score quite huge, but the ‘label_encoded_preciplevel’ feature, let the R^2 score drop down a bit compared, which means this feature in this data frame doesn’t provide positive impact on model predictions. Therefore, I finally decided to remove this feature, therefore the final selected features are: ['YEAR', 'MONTH', 'DAY', 'WEEKDAY', 'HOUR'].

B. Machine Learning Methodology

This task is a regression problem and aimed to predict the bike usage. I have used **Random Forest Regressor**, **SVM** and **K-Nearest Neighbors Regressor** and after compare the performance of each model on this task, **Random Forest Regressor** model is my final choice for this task.

- **Random Forest Regressor**

  Random Forest Regressor model is a model that combined multiple decision trees or multiple models. Since this task is a regression task, therefore each leaf nodes of decision tree represent numerical values instead of classes. In this model, each decision tree model will take a random subset of the training data and trained, after trained each decision tree will predict a value for the given data. And the overall performance of the random forest regressor model is the average of all the individual decision trees’ predictions.

  There are several parameters that can be used to optimize its performance, for example **max_depth**, it determines the max depth of the decision trees. **max_features**, which set the maximum number of features the model will consider. In this task, we keep these parameters as default. The most important parameter of this model is:

  n_estimators: this parameter defined the number of decision trees you want to include in this model. Higher value, higher performance, but may lead to a more complex model that led data overfitting. In this task, we will try to find the best n_estimators value.

  Loss Function: Mean Squared error is the most common cost function for the random forest regressor model, this model tries to minimize the value of mean squared error, which is the average squared difference between the estimated values and the actual value.

- **SVM**

  Support Vector Regression model is derived from SVM and used mainly for regression tasks. It predicts a continuous output. The primary idea behind SVR is to create a function that find the relationship between the training features and the continuous predict target variable within a certain threshold. There are several parameters in this model:

  kernel: It is used to transform the data into a higher-dimensional space. SVM can efficiently conduct non-linear classification using a linear classifier by employing the kernel method.

  C: This is a regularization parameter; it controls the tolerance for errors. It is a very important parameter, the choice of values requires careful consideration, with a higher C value, the model become less tolerance for errors, which means the model try to fit every point into the model, that may cause overfitting problem. With a lower C value, the model is more tolerant of errors, which may cause the model too simple to predict and leading to underfitting problem.

  Epsilon (ε): Sets the width of the ε-insensitive zone. A smaller ε means the model is less tolerant of errors, while a larger ε makes the model more generalized. One of the uses of this parameter is to control the C penalty, point fall within this zone will not be penalized in loss function.

  Loss Function: epsilon-insensitive loss function, which means point fall within the epsilon-insensitive zone will not be penalty, and the calculation is : **L(y, f(x)) = max(0, | y – f(x) |- ε)** where **y** is the actual value(features), **f(x)** is the predicted value , **ε** is predefined margin of tolerance.

- **K-Nearest Neighbors**

  KNN is a common model used in machine learning area. It can be used for both classification task and regression task. KNN calculate the distance between the current point and other points in the training set by using Euclidean or Manhattan. Then this model will review the value of K and select the nearest point to the input data. Then when KNN predicts the target value, it will be averaging the values of K nearest neighbors of the input data.
  Here below are some commonly used parameters and hyperparameters of the KNN:

  n_neighbors: number of the neighbors, it is the most important hyperparameter, when initialized the KNN model, this hyperparameter need to be set manually. A smaller value of K will lead to higher influence of the noise data. A larger value of K may include the data point that too far away from current data point. Therefore, the choice of the value of K need to be evaluated carefully in order to select a good K.

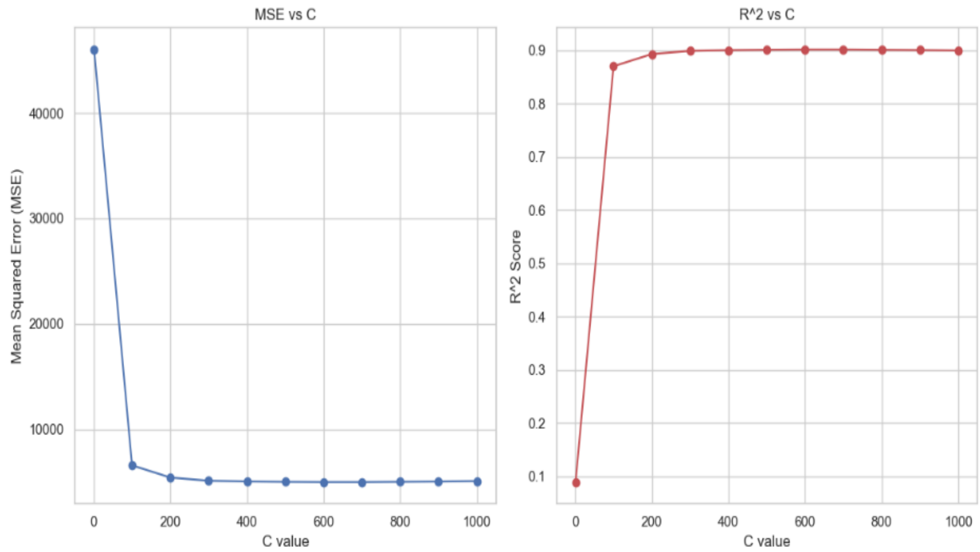
  metric: The formula that used to calculate the similarity, like Euclidean Distance, Manhattan Distance.

  weights: Used to give more importance to its neighbors to get a better predication result. Normally, it set as default ‘uniform’

C. Models Evaluation

To evaluate the performance of the three chosen models and decided which one is the final chosen one. The first step involves tuning the parameters of the three models themselves to be optimal for the current dataset. To fine-tunning each model, different models will adjust different parameters. R^2 score and mean squared error will be used to evaluate the performance.

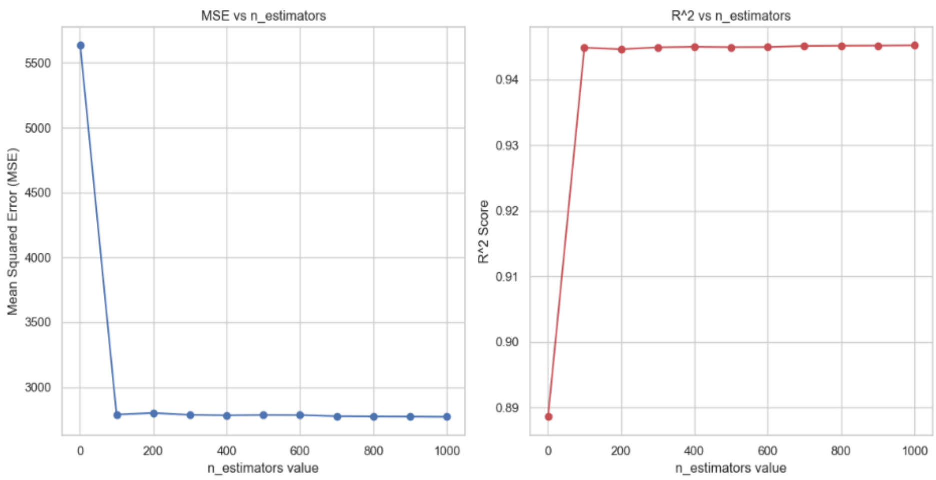
For SVM model, since C is the most important parameter, I will adjust the value of C in range [1,100,200,300,400,500,600,700,800,900,1000] to find the best C with the best R^2 score and MSE.



	C_value	MSE	R2
0	1	46038.876370	0.089537
1	100	6570.980329	0.870053
2	200	5416.210862	0.892889
3	300	5108.582492	0.898973
4	400	5044.523497	0.900240
5	500	5005.353194	0.901014
6	600	4983.383217	0.901449
7	700	4982.433912	0.901468
8	800	5008.301194	0.900956
9	900	5037.108759	0.900386
10	1000	5075.585036	0.899625

According to the line graphs of MSE vs. C and R^2 vs. C shown by the models trained with different values of C, we can find that when c = 200, as the value of c increases, the values of MSE and R^2 have stabilized and are not much different from the value when c is 200, the larger the C, the more complex the model, which will lead to the overfitting problem in the model, so in order to avoid this problem, we prioritize the smallest value of C when the scores are not much different, and therefore **200** is the best value of C in the model of this SVM. Therefore, for SVM, when **C = 200, R^2 = 0.8928, MSE = 5416.21**

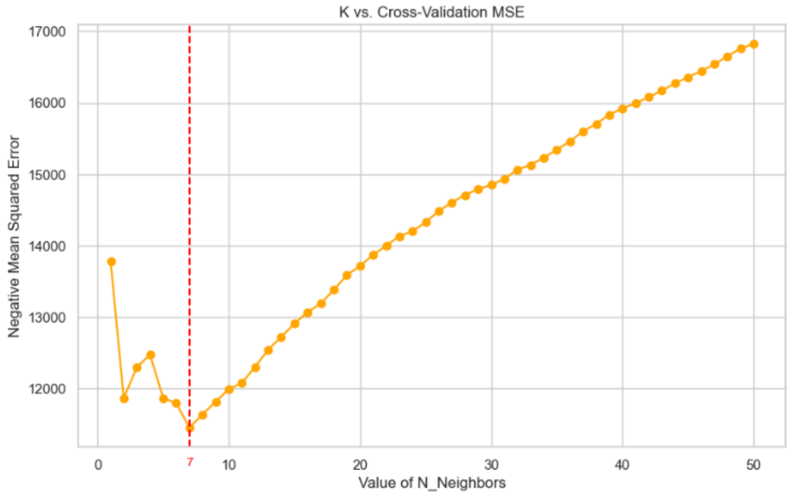
For Random Forest Model, n_estimators is the most important parameters, I will also adjust the value of n_estimators in range [1,100,200,300,400,500,600,700,800,900,1000] and follow the same fine-tunning steps as SVM.



	n_estimators_value	MSE	R2
0	1	5634.280576	0.888577
1	100	2789.522474	0.944835
2	200	2801.400048	0.944600
3	300	2787.106165	0.944882
4	400	2782.887800	0.944966
5	500	2785.654477	0.944911
6	600	2784.858740	0.944927
7	700	2776.481739	0.945092
8	800	2774.495711	0.945132
9	900	2773.669623	0.945148
10	1000	2771.855008	0.945184

From the above plots, similar to SVM, 200 is the best choice. Therefore, when **n_estimators = 200**, **R^2 = 0.944, MSE = 2801.40**

For KNN model, the fine-tunning process will focus on find out the best value of K(N_Neighbors). The rest parameters set as default. This time, unlike the fine-tunning process of the SVM and Random Forest. It is not a good idea to set a very large value of K, therefore, instead of setting range up to 1000, we will set the range up to 51. Here below is the plot that display the best value of K.

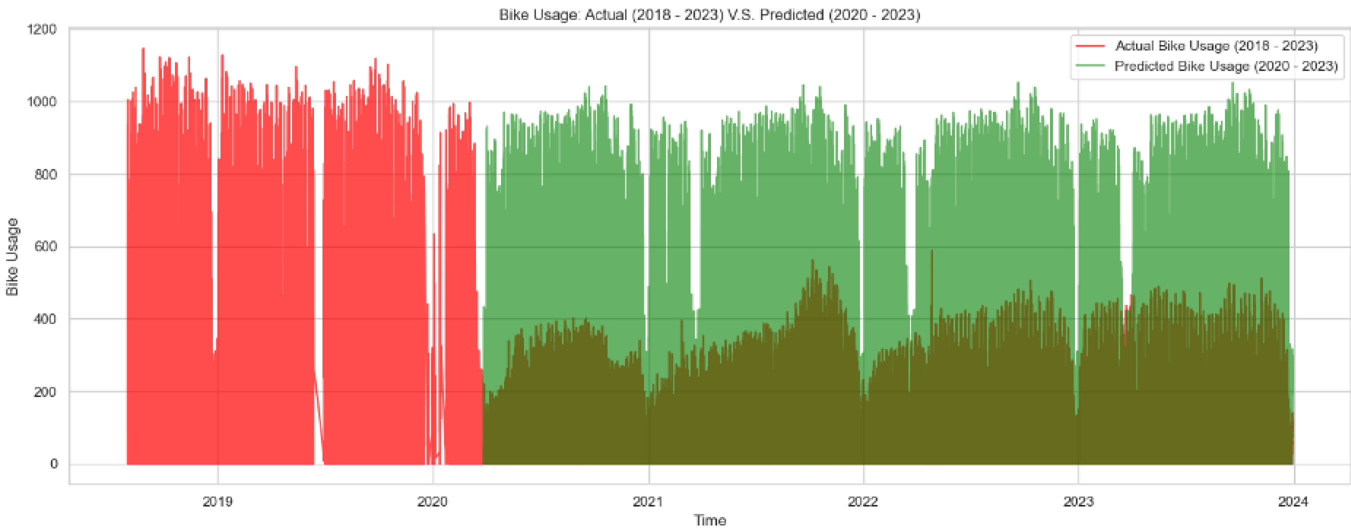


From the plot, we can see that when k = 7, the value of MSE is the lowest, and when K is larger than 7, the value of MSE increase very fast. Therefore, when **k = 7** is the best value. **R^2 = 0.8586**, **MSE = 11455.97**

Finally, compared to the R^2 and MSE, we can see that the Random Forest Model is the best model with around **0.944 R^2 score .** Therefore, the final model used to predict is Random Forest Model.

D. Prediction Evaluation

After using the Random Forest Model to predict the bike usage start from 2020-03-27 to 2023-12-30, here below is the prediction plots.

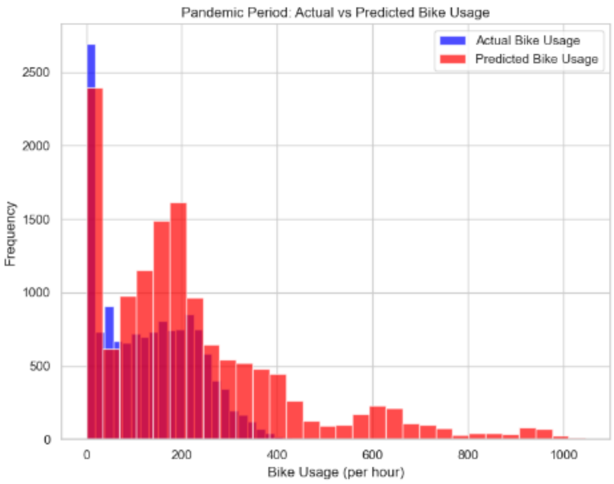


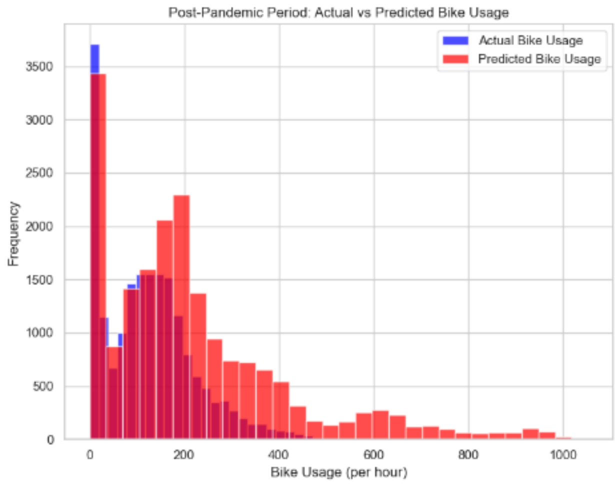
From this complete prediction chart, the green portion predicts the number of bikes that would have been used during this period if the epidemic had not occurred, and we can see that the predicted portion in green has the same trend compared to the actual portion in red, indicating that the model did a good job of predicting. From this graph, we can visualize that the predicted bicycle usage is much higher than the actual usage, which shows the serious impact of the epidemic on the usage of public bicycles.

With a detailed tally of predicted average hourly bicycle use versus actual average hourly bike usage during and after the pandemic, and a allows us to draw more detailed conclusions, here below is the report:

	Pandemic_Hourly_Usage	Pandemic_Predicted_Hourly_Usage	Post_Pandemic_Hourly_Usage	Post_Pandemic_Predicted_Hourly_Usage
mean	129.90	222.13	123.13	217.37
std	101.23	202.79	97.41	199.14
min	0.00	0.00	0.00	0.00
25%	37.00	82.00	37.00	79.00
50%	126.00	179.00	117.00	177.00
75%	210.00	300.00	178.00	291.25
max	562.00	1046.00	589.00	1053.00

From the report, we can find that the average usage during the pandemic was **41% lower** than the predicted average, **54.87% lower** in the first quartile of the overall data, **29.6% lower** in the second quartile, and **30% lower** in the third quartile, Similar to the after-pandemic period, average usage was 43% lower than the predicted, 53.16% **lower** in the first quartile of the overall data, **33.89% lower** in the second quartile, and **38.88%% lower** in the third quartile. And combine with the below histogram of the overall frequency of hourly bicycle usage during two periods, shows that most of the use is still concentrated around 200, indicating that people's habits have been changed by the pandemic and indicating that bike usage was impacted throughout the full pandemic period and recovered slowly. This is not a short-term recovery process.





In conclusion, the impact of the pandemic on the city bike usage for both the pandemic period and the post-pandemic period are very serious, the company should reduce the related capital investment.

Part 2

(i) ROC curve is a plot that used to evaluate the performance of a classification machine learning model. It measures the True Positive Rate against the False Positive Rate at various threshold value. In confusion matrix, True positive is the model predicted positive and the actual class is positive.
True Positive Rate = TP /TP+FN. (TP+FN) represents the total number of all samples in the model that are positive classes. Normally this value is the recall value of the model.
False Positive Rate = FP /(FP+TN). FP is the number of negative instances incorrectly predicted as positive. (FP+TN) represents the total number of all samples in the model that are negative classes. Therefore, ROC curve is a line moves from the bottom left corner to the upper right corner. To compare the performance of a classifier with a baseline classifier, we can see the curves on the plot. The curve of a baseline classifier normally a diagonal line. The curve for a good classifier is normally a very steep curve from the left bottom corner, therefore, the farther this curve is from the curve of the baseline model, the better this classifier performs. We can also use the area under the curve(AUC) to compare the performance. The higher the area, the higher the performance. Using ROC instead of the classification accuracy metric is because ROC curve evaluate the performance of a model without need to worry about the value of threshold, it is more general. And it perform

well in imbalanced dataset.

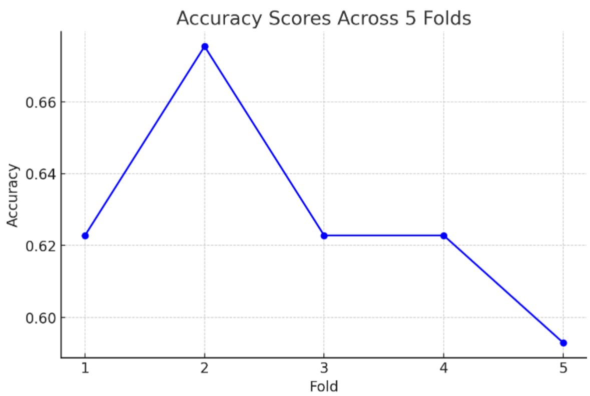
(ii) **The first example** is when the relationship between the features and predict target is a **non-linear relationship**, in this case, linear regression will not be able to capture the trend, therefore provide inaccurate predictions. The reason is that the linear model is too simple to capture the complex features. To solve this situation, we can try a new model that can capture the non-linear relationships. For example, any complex regression model, like random forest regressor or ridge regression model.

The second example is the dataset **presence of many outliers**. Outliers will influence the linear regression a lot, it may cause the linear regression model to fail to accurately find the regression line and fail to compute accurate coefficients and intercepts because it will make the data more volatile. To solve this situation, you need to reprocess the dataset to eliminate outliers. Another way is to use ridge regression or lasso regression, they can reduce the impact of outliers due to its unique regularization property, like L1 regularization.

(iii) **kernel in SVM:** It is used to transform the data into a higher-dimensional space. When the data is not linearly separable, SVM employs kernels. SVM can efficiently conduct non-linear classification using a linear classifier by employing the kernel method. It will be helpful for SVM to handle non-linear data or more complex dataset.
Types of kernels in SVM: Linear Kernel, Polynomial Kernel, Gaussian Kernel, and Sigmoid Kernel.

kernel in CNNs: It is a small matrix act as filter and used to extract features from input data. The kernel is like a slide window on input data, it recalculates the data it covered and output a new value, and it will continue slide to next part of the input data, and finally the new data will generate a new matrix, this process also called convolution. For example, input data is a matrix [[1,2,3],[4,5,6],[7,8,9]], a kernel is [[1,0],[0.-1]], after convolution, the output matrix is: [[-4,-4][-4,-4]]. Kernels in CNNs are used when handle image process tasks, it can help to extract features from the image or reduce thee dimensions of the images.

(iv) The idea behind the multiple resampling is to train and test the model on different subsets of the data. It will provide a general and reasonable performance score of the model because when trained the model on a single fixed split portion of the dataset, we don't know if the model performs well in other unseen portion datasets as well. Resampling multiple times can help us to evaluate the performance of the model on unseen data. For example, I used a classification dataset and applied SVC model to train the data and use 5-fold cross validation to test the accuracy, as shown in the below plot, 5 rounds provide five different accuracy scores, which means it randomly splitting the training dataset 5 times. Therefore, we can get a average accuracy score if the model, which is more general and reliable.



When the dataset is small and you need a more reliable model, it is appropriate to apply K-fold cross validation, because it provides you different combinations of training and test sets.
When the dataset is too large, then maybe one split of the training set and test set will be enough for model to train and provide a good performance, it is then not appropriate time to apply K-fold cross validation. And when the dataset is time-series data, it is not appropriate time to apply K-fold cross validation because random splitting may result in destroying the invisible patterns present in the data, leading to model training failures.

APPENDIX

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import seaborn as sns
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score, recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import mean_squared_error

## Data Preprocessing and Feature Engineering
file_paths_2018 = ['./2018/dublinbikes_20180701_20181001.csv',
                   './2018/dublinbikes_20181001_20190101.csv']

file_paths_2019 = ['./2019/dublinbikes_20190101_20190401.csv',
                   './2019/dublinbikes_20190401_20190701.csv',
                   './2019/dublinbikes_20190701_20191001.csv',
                   './2019/dublinbikes_20191001_20200101.csv']

file_paths_2020 = ['./2020/dublinbikes_20200101_20200401.csv',
                   './2020/dublinbikes_20200401_20200701.csv',
                   './2020/dublinbikes_20200701_20201001.csv',
                   './2020/dublinbikes_20201001_20210101.csv']

file_paths_2021 = ['./2021/dublinbikes_20210101_20210401.csv',
                   './2021/dublinbikes_20210401_20210701.csv',
                   './2021/dublinbikes_20210701_20211001.csv',
                   './2021/dublinbikes_20211001_20220101.csv']

file_paths_2022 = ['./2022/dublinbike-historical-data-2022-01.csv',
                   './2022/dublinbike-historical-data-2022-02.csv',
                   './2022/dublinbike-historical-data-2022-03.csv',
                   './2022/dublinbike-historical-data-2022-04.csv',
                   './2022/dublinbike-historical-data-2022-05.csv',
                   './2022/dublinbike-historical-data-2022-06.csv',
                   './2022/dublinbike-historical-data-2022-07.csv',
                   './2022/dublinbike-historical-data-2022-08.csv',
                   './2022/dublinbike-historical-data-2022-09.csv',
                   './2022/dublinbike-historical-data-2022-10.csv',
                   './2022/dublinbike-historical-data-2022-11.csv',
                   './2022/dublinbike-historical-data-2022-12.csv']

file_paths_2023 = ['./2023/dublinbike-historical-data-2023-01.csv',
                   './2023/dublinbike-historical-data-2023-02.csv',
                   './2023/dublinbike-historical-data-2023-03.csv',
                   './2023/dublinbike-historical-data-2023-04.csv',
                   './2023/dublinbike-historical-data-2023-05.csv',
```

```
                    './2023/dublinbike-historical-data-2023-06.csv',
                    './2023/dublinbike-historical-data-2023-07.csv',
                    './2023/dublinbike-historical-data-2023-08.csv',
                    './2023/dublinbike-historical-data-2023-09.csv',
                    './2023/dublinbike-historical-data-2023-10.csv',
                    './2023/dublinbike-historical-data-2023-11.csv',
                    './2023/dublinbike-historical-data-2023-12.csv']

def load_and_concatenate_csv(file_paths):
    all_data = [pd.read_csv(file) for file in file_paths]
    return pd.concat(all_data, ignore_index=True)

df_2018 = load_and_concatenate_csv(file_paths_2018)
df_2019 = load_and_concatenate_csv(file_paths_2019)
df_2020 = load_and_concatenate_csv(file_paths_2020)
df_2021 = load_and_concatenate_csv(file_paths_2021)
df_2022 = load_and_concatenate_csv(file_paths_2022)
df_2023 = load_and_concatenate_csv(file_paths_2023)

# Create a dataframe dictionary include all years of data
dataframes = {
    2018: df_2018,
    2019: df_2019,
    2020: df_2020,
    2021: df_2021,
    2022: df_2022,
    2023: df_2023
}

summary = []

# Iterate through the dictionary and count the number of rows and columns in each DataFrame
for year, df in dataframes.items():
    rows, columns = df.shape
    summary.append([year, rows, columns])

# Convert aggregated information to DataFram
summary_df = pd.DataFrame(summary, columns=['Year', 'Rows', 'Columns'])


summary_df

for year, df in dataframes.items():
    # Change Time column to datetime format
    df['TIME'] = pd.to_datetime(df['TIME'])
    df['LAST UPDATED'] = pd.to_datetime(df['LAST UPDATED'])
    df.rename(columns={'BIKE STANDS': 'BIKE_STANDS'}, inplace=True)
    df.rename(columns={'AVAILABLE BIKE STANDS': 'AVAILABLE_BIKE_STANDS'}, inplace=True)
    df.rename(columns={'AVAILABLE BIKES': 'AVAILABLE_BIKES'}, inplace=True)
```

```
    df['DATE'] = df['TIME'].dt.date
    df['WEEKDAY'] = df['TIME'].dt.weekday
    df['YEAR'] = df['TIME'].dt.year
    df['MONTH'] = df['TIME'].dt.month
    df['DAY'] = df['TIME'].dt.day
    df['HOUR'] = df['TIME'].dt.hour

missing_summary_df = pd.DataFrame()

for year, df in dataframes.items():
    print(f"Head of the {year} DataFrame:")
    print(df.head())
    print("\n")
    # Check missing value
    missing_values = df.isnull().sum()

    missing_summary_df[year] = missing_values

    # Check datatypes of each dataframe
    print(df.dtypes)

# display the results
missing_summary_df
df_2023.shape

# Looping through the dictionary and printing the head of each dataframe
for year, df in dataframes.items():
    print(f"Head of the {year} DataFrame:")
    print(df.head())    # Notice the parentheses here
    print("\n")
# 2020 Non-Pandemic period 1.1 to 3.27
df_2020_nonPandemic = df_2020[(df_2020['TIME'] >= pd.to_datetime('2020-01-01')) & (df_2020['TIME'] <= pd.to_datetime('2020-03-27'))]
# 2020 Pandemic period 3.27 to 12.31
df_2020_Pandemic = df_2020[df_2020['TIME'] > pd.to_datetime('2020-03-27')]

# 2021 Pandemic period 1.1 to 10.22
df_2021_Pandemic = df_2021[(df_2021['TIME'] >= pd.to_datetime('2021-01-01')) & (df_2021['TIME'] <= pd.to_datetime('2021-10-22'))]
# 2021 Non-Pandemic period 10.22 to 12.31
df_2021_nonPandemic = df_2021[df_2021['TIME'] > pd.to_datetime('2021-10-22')]

#Combine different dataframes based on Pandemic timeline
df_beforePandemic = pd.concat([df_2018, df_2019,df_2020_nonPandemic])
df_Pandemic = pd.concat([df_2020_Pandemic, df_2021_Pandemic])
df_afterPandemic = pd.concat([df_2021_nonPandemic, df_2022, df_2023])
df_allData = pd.concat([df_beforePandemic, df_Pandemic, df_afterPandemic])

#Drop any NA values
```

```
df_beforePandemic = df_beforePandemic.dropna()
df_Pandemic = df_Pandemic.dropna()
df_afterPandemic = df_afterPandemic.dropna()
df_allData = df_allData.dropna()

#Sort the dataset
df_beforePandemic = df_beforePandemic.sort_values(by=['STATION ID', 'TIME'])
df_Pandemic = df_Pandemic.sort_values(by=['STATION ID', 'TIME'])
df_afterPandemic = df_afterPandemic.sort_values(by=['STATION ID', 'TIME'])
df_allData = df_allData.sort_values(by=['STATION ID', 'TIME'])

# Calculate the Bike usage
# Shift the 'AVAILABLE_BIKES' column down by 1 to align the nth row with the (n+1)th row for subtraction
df_allData['PREV_AVAILABLE_BIKES'] = df_allData.groupby(['STATION ID',
'DATE'])['AVAILABLE_BIKES'].shift(1)

# Calculate the difference and replace negative values with 0
# Calculate the usage as (nth row - (n+1)th row)
df_allData['BIKE_USAGE'] = df_allData['PREV_AVAILABLE_BIKES'] - df_allData['AVAILABLE_BIKES']
df_allData['BIKE_USAGE'] = df_allData['BIKE_USAGE'].apply(lambda x: 0 if x < 0 else x)

# Fill NA values with 0 in the 'USAGE' column
df_allData['BIKE_USAGE'] = df_allData['BIKE_USAGE'].fillna(0)

df_allData['BIKE_USAGE'] = df_allData['BIKE_USAGE'].astype(int)

# Remove the 'PREV_AVAILABLE_BIKES' column as it is no longer needed
df_allData.drop(columns=['PREV_AVAILABLE_BIKES'], inplace=True)

# Display the first few rows to verify the changes
df_allData.head()

#Calculate the Bike usage for beforePandemic subset that used for train model

df_beforePandemic['PREV_AVAILABLE_BIKES'] = df_beforePandemic.groupby(['STATION ID',
'DATE'])['AVAILABLE_BIKES'].shift(1)

# Calculate the difference and replace negative values with 0
# Calculate the usage as (nth row - (n+1)th row)
df_beforePandemic['BIKE_USAGE'] = df_beforePandemic['PREV_AVAILABLE_BIKES'] -
df_beforePandemic['AVAILABLE_BIKES']
df_beforePandemic['BIKE_USAGE'] = df_beforePandemic['BIKE_USAGE'].apply(lambda x: 0 if x < 0 else x)

# Fill NA values with 0 in the 'USAGE' column
df_beforePandemic['BIKE_USAGE'] = df_beforePandemic['BIKE_USAGE'].fillna(0)

df_beforePandemic['BIKE_USAGE'] = df_beforePandemic['BIKE_USAGE'].astype(int)

# Remove the 'PREV_AVAILABLE_BIKES' column as it is no longer needed
```

```
df_beforePandemic.drop(columns=['PREV_AVAILABLE_BIKES'], inplace=True)

# Display the first few rows to verify the changes
df_beforePandemic.head()

#Aggregate beforePandemic data by hour
df_beforePandemic_hourly_bike_usage = df_beforePandemic.groupby(['YEAR','MONTH','DAY',
'WEEKDAY','HOUR'])['BIKE_USAGE'].sum().reset_index()
df_beforePandemic_hourly_bike_usage['TIME'] = pd.to_datetime(df_beforePandemic_hourly_bike_usage[['YEAR',
'MONTH', 'DAY', 'HOUR']]).dt.strftime('%Y-%m-%d %H:00:00')

# Aggregate all data by hour
df_allData_hourly_bike_usage = df_allData.groupby(['YEAR','MONTH','DAY',
'WEEKDAY','HOUR'])['BIKE_USAGE'].sum().reset_index()
df_allData_hourly_bike_usage['TIME'] = pd.to_datetime(df_allData_hourly_bike_usage[['YEAR', 'MONTH', 'DAY',
'HOUR']]).dt.strftime('%Y-%m-%d %H:00:00')

#Change the Time datatype to datetime
df_beforePandemic_hourly_bike_usage['TIME'] = pd.to_datetime(df_beforePandemic_hourly_bike_usage['TIME'])
df_allData_hourly_bike_usage['TIME'] = pd.to_datetime(df_allData_hourly_bike_usage['TIME'])

# Define the time periods
periods = {
    'Before_pandemic': ('2018-08-01', '2020-03-26'),
    'Pandemic': ('2020-03-27', '2021-10-21'),
    'After_pandemic': ('2021-10-22', '2023-12-30')
}

# Calculate the total number of days for each phase
def calculate_days(start_date, end_date):
    start_date = pd.to_datetime(start_date)
    end_date = pd.to_datetime(end_date)
    return (end_date - start_date).days + 1

# Display Descriptive Statistical Data report
def calculate_statistics(data, start_date, end_date):
    period_data = data[(data['TIME'] >= start_date) & (data['TIME'] <= end_date)]
    total_days = calculate_days(start_date, end_date)
    total_usage = period_data['BIKE_USAGE'].sum()
    average_daily_usage = total_usage / total_days
    average_hourly_usage = period_data['BIKE_USAGE'].mean()
    daily_usage = period_data.groupby(period_data['TIME'].dt.date)['BIKE_USAGE'].sum()
    weekly_usage = period_data.groupby(period_data['TIME'].dt.weekday)['BIKE_USAGE'].sum()
    monthly_usage = period_data.groupby(period_data['TIME'].dt.month)['BIKE_USAGE'].sum()
    median = period_data['BIKE_USAGE'].median()
    std_dev = period_data['BIKE_USAGE'].std()


    return {
```

```
        'total_days': total_days,
        'total_usage': total_usage,
        'average_daily_usage': average_daily_usage,
        'average_hourly_usage': average_hourly_usage,
        'daily_usage': daily_usage,
        'weekly_usage': weekly_usage,
        'monthly_usage': monthly_usage,
        'median': median,
        'std_dev': std_dev
    }

# Calculate statistics for each period
stats = {period: calculate_statistics(df_allData_hourly_bike_usage, pd.to_datetime(start), pd.to_datetime(end))
         for period, (start, end) in periods.items()}

key_stats = {}
for period, data in stats.items():
    key_stats[period] = {
        'Total_usage': data['total_usage'],
        'Average_hourly_usage': data['average_hourly_usage'],
        'Max_daily_usage': data['daily_usage'].max(),
        'Min_daily_usage': data['daily_usage'].min(),
        'Average_daily_usage': data['average_daily_usage'],
        'Average_weekly_usage': data['weekly_usage'].mean(),
        'Average_monthly_usage': data['monthly_usage'].mean(),
        'Median': data['median'],
        'Standard Deviation': data['std_dev']
    }

# Formatting the key statistics for easier reading
ThreePeriods_stats_report = pd.DataFrame(key_stats).map(lambda x: "{:.2f}".format(x) if isinstance(x, float) else x)

ThreePeriods_stats_report

# Set the style of seaborn
sns.set(style="whitegrid")

# Function to plot data
def plot_data(data, title, ylabel, xlabel='Date'):
    plt.figure(figsize=(12, 6))
    plt.plot(data)
    plt.title(title)
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)
    plt.show()

# Plotting the daily bike usage for each period
for period, data in stats.items():
    plot_data(data['daily_usage'], f'Daily Bike Usage - {period.replace("_", " ").title()}', 'Bike Usage')
```

```
df_beforePandemic_hourly_bike_usage.head()

### Add weather data to the dataset
df_weather = pd.read_csv('DublinWeather.csv')
df_weather.head(2)

# Define a function to categorize precipitation levels
def categorize_precipitation(precip):
    if precip == 0:
        return 'No Rain'
    elif 0 < precip <= 2.5:
        return 'Light Rain'
    elif 2.5 < precip <= 7.6:
        return 'Moderate Rain'
    elif 7.6 < precip <= 50:
        return 'Heavy Rain'
    else:
        return 'Violent Rain'

# Apply the function to the 'precip' column to create the 'precip_level' column
df_weather['precip_level'] = df_weather['precip'].apply(categorize_precipitation)

from sklearn.preprocessing import LabelEncoder

# Creating the label encoder
label_encoder = LabelEncoder()

# Applying label encoding to the 'preciptype' column
df_weather['label_encoded_preciplevel'] = label_encoder.fit_transform(df_weather['precip_level'])

# Display the first few rows of the updated DataFrame
df_weather.head(50)

# Get the category and its corresponding code
classes = label_encoder.classes_
encoded_values = range(len(classes))

encoding_dict = dict(zip(classes, encoded_values))

encoding_dict

df_weather_copy = df_weather[['datetime', 'precip', 'precip_level', 'label_encoded_preciplevel']].copy()
df_weather_copy['year'] = pd.to_datetime(df_weather_copy['datetime']).dt.year
df_weather_copy['month'] = pd.to_datetime(df_weather_copy['datetime']).dt.month
df_weather_copy['day'] = pd.to_datetime(df_weather_copy['datetime']).dt.day
df_weather_copy['hour'] = pd.to_datetime(df_weather_copy['datetime']).dt.hour

# Remove duplicate rows in weather_data based on columns 'year', 'month', 'day', and 'hour'
df_weather_copy = df_weather_copy.drop_duplicates(subset=['year', 'month', 'day', 'hour'], keep='first')
```

```
df_weather_copy.head()

#Merge weather data to beforePandemic_hourly dataframe
df_beforePandemic_hourly_withWeather = pd.merge(df_beforePandemic_hourly_bike_usage, df_weather_copy, left_on=['YEAR', 'MONTH', 'DAY', 'HOUR'],
                                right_on=['year', 'month', 'day', 'hour'], how='left')

# Keep only the necessary columns from the merged dataframe
df_beforePandemic_hourly_withWeather = df_beforePandemic_hourly_withWeather[['YEAR', 'MONTH', 'DAY', 'HOUR', 'WEEKDAY', 'BIKE_USAGE', 'TIME', 'precip', 'precip_level', 'label_encoded_preciplevel']]

# Drop rows where 'label_encoded_preciplevel' is NaN
df_beforePandemic_hourly_withWeather = df_beforePandemic_hourly_withWeather.dropna()

df_beforePandemic_hourly_withWeather['label_encoded_preciplevel'] = df_beforePandemic_hourly_withWeather['label_encoded_preciplevel'].astype(int)

df_beforePandemic_hourly_withWeather

df_beforePandemic_hourly_withWeather.shape

df_beforePandemic_hourly_bike_usage.shape

## Features selection

from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
import datetime as dt

# Splitting the dataset into features (X) and target (y)
X = df_beforePandemic_hourly_withWeather[['YEAR', 'MONTH', 'DAY','HOUR']]
y = df_beforePandemic_hourly_withWeather['BIKE_USAGE']

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Training the Random Forest model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Checking model's performance on the test set
rf_model.score(X_test, y_test)

# Include preciplevel and Weekday features

# Splitting the dataset into features (X) and target (y)
X = df_beforePandemic_hourly_withWeather[['YEAR', 'MONTH', 'DAY', 'WEEKDAY', 'HOUR','label_encoded_preciplevel']]
```

```
y = df_beforePandemic_hourly_withWeather['BIKE_USAGE']

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,random_state=42)

# Training the Random Forest model
rf_model = RandomForestRegressor(n_estimators=100,random_state=42)
rf_model.fit(X_train, y_train)

# Checking model's performance on the test set
rf_model.score(X_test, y_test)

# Include WEEKDAY feature
# Splitting the dataset into features (X) and target (y)
X = df_beforePandemic_hourly_withWeather[['YEAR', 'MONTH', 'DAY', 'WEEKDAY', 'HOUR']]
y = df_beforePandemic_hourly_withWeather['BIKE_USAGE']

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Training the Random Forest model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Checking model's performance on the test set
rf_model.score(X_test, y_test)
```

Final feature selection: ['YEAR', 'MONTH', 'DAY', 'WEEKDAY', 'HOUR'] Predict feature: 'BIKE_USAGE'

```
# Final training set by Dropping the specified not used columns
df_beforePandemic_FinalTrainSet = df_beforePandemic_hourly_withWeather.drop(columns=['precip', 'precip_level',
'label_encoded_preciplevel'])
df_beforePandemic_FinalTrainSet
```

Try 3 different machine learning methodologies to evaluate the performance
Fine-tuning SVM model

```
from sklearn.svm import SVR
# Include WEEKDAY feature
# Splitting the dataset into features (X) and target (y)
X = df_beforePandemic_FinalTrainSet[['YEAR', 'MONTH', 'DAY', 'WEEKDAY', 'HOUR']]
y = df_beforePandemic_FinalTrainSet['BIKE_USAGE']

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Find a good C value
def SVM_C_eval(C):
    svm_reg = SVR(gamma='auto',C=C)

    # Model Train
    svm_reg.fit(X_train,y_train)

    # Prediction
    y_predict_svm = svm_reg.predict(X_test)

    # Evaluation
    svm_mse = mean_squared_error(y_test,y_predict_svm)

    # mean_squared_error

    return svm_mse, svm_reg.score(X_test,y_test)


C_value = [1,100,200,300,400,500,600,700,800,900,1000]
svm_mse_list = []
r2_list = []

for c in C_value:
    svm_mse, r2 = SVM_C_eval(c)
    print(f'MSE of SVM Regressor:  {svm_mse}')
    print(f'R^2 Score of SVM Regressor:  {r2}')
    svm_mse_list.append(svm_mse)
    r2_list.append(r2)
    print('----------------------------------')

# Draw two subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# MSE vs C
ax1.plot(C_value, svm_mse_list, 'b-o')
ax1.set_title('MSE vs C')
ax1.set_xlabel('C value')
ax1.set_ylabel('Mean Squared Error (MSE)')
ax1.grid(True)

# R^2 vs C
ax2.plot(C_value, r2_list, 'r-o')
ax2.set_title('R^2 vs C')
ax2.set_xlabel('C value')
ax2.set_ylabel('R^2 Score')
ax2.grid(True)

plt.tight_layout()
plt.show()
```

```
SVM_evaluation_data = {
    'C_value': list(C_value),
    'MSE': svm_mse_list,
    'R2': r2_list
}
df_SVM_evaluation = pd.DataFrame(SVM_evaluation_data)
df_SVM_evaluation

## Fine Tunning Random Forest Model

def RandomForest_n_estimators_eval(n):
    # Training the Random Forest model
    rf_model_eval = RandomForestRegressor(n_estimators = n, random_state=42)
    rf_model_eval.fit(X_train, y_train)


    # Prediction
    y_predict_rf = rf_model_eval.predict(X_test)

    # Evaluation
    rf_mse = mean_squared_error(y_test,y_predict_rf)


    return rf_mse, rf_model_eval.score(X_test,y_test)

estimators_value = [1,100,200,300,400,500,600,700,800,900,1000]
rf_mse_list = []
rf_r2_list = []

for value in estimators_value:
    rf_mse, rf_r2 = RandomForest_n_estimators_eval(value)
    print(f'MSE of Random Forest: {rf_mse}')
    print(f'R^2 score of Random Forest: {rf_r2}')
    rf_mse_list.append(rf_mse)
    rf_r2_list.append(rf_r2)
    print('----------------------------------')

# Draw two sub plots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# MSE vs C
ax1.plot(estimators_value, rf_mse_list, 'b-o')
ax1.set_title('MSE vs n_estimators')
ax1.set_xlabel('n_estimators value')
ax1.set_ylabel('Mean Squared Error (MSE)')
ax1.grid(True)

# R^2 vs C
```

```
ax2.plot(estimators_value, rf_r2_list, 'r-o')
ax2.set_title('R^2 vs n_estimators')
ax2.set_xlabel('n_estimators value')
ax2.set_ylabel('R^2 Score')
ax2.grid(True)

plt.tight_layout()
plt.show()

rf_evaluation_data = {
    'n_estimators_value': list(estimators_value),
    'MSE': rf_mse_list,
    'R2': rf_r2_list
}
df_rf_evaluation = pd.DataFrame(rf_evaluation_data)
df_rf_evaluation
```

KNN

```
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsRegressor
import matplotlib.pyplot as plt

def choose_k_knn(X, y, k_range, plot_color):
    ''' Function to determine optimal k for KNN using cross-validation '''
    cv_scores = []

    # Cross-validation for each k value
    for k in k_range:
        knn = KNeighborsRegressor(n_neighbors=k)
        # Using negative mean squared error for scoring
        scores = cross_val_score(knn, X, y, cv=5, scoring='neg_mean_squared_error')
        cv_scores.append(scores.mean())

    # Convert to positive mean squared error
    mse_scores = [-x for x in cv_scores]

    # Determine the optimal k value
    optimal_k = k_range[np.argmin(mse_scores)]
    print(f"Optimal K: {optimal_k}, MSE: {-max(cv_scores)}")

    # Plotting the cross-validation scores
    plt.figure(figsize=(10, 6))
    plt.plot(k_range, mse_scores, marker='o', linestyle='-', color=plot_color)
    plt.xlabel('Value of N_Neighbors')
    plt.ylabel('Negative Mean Squared Error')
    plt.title('K vs. Cross-Validation MSE')
    plt.axvline(x=optimal_k, color='red', linestyle='--')
```

```
    y_position = plt.ylim()[0] - 0.052 * (plt.ylim()[1] - plt.ylim()[0])
    plt.text(optimal_k, y_position, f'{optimal_k}', color='red', ha='center', va='bottom', fontsize=10)

    plt.grid(True)
    plt.show()

# Implement
k_range = list(range(1, 51))
plot_color = 'orange'
choose_k_knn(X, y, k_range, plot_color)


# Get the R^2 Score for the best knn model
# Initialize KNN regressor
knn_regressor = KNeighborsRegressor(n_neighbors=7)

# Train the model
knn_regressor.fit(X_train, y_train)

# Predict on the test set
y_pred_KNN = knn_regressor.predict(X_test)
r2_score = knn_regressor.score(X_test,y_test)

# Evaluate the model
print('KNN R^2 Score: ',r2_score)

## Machine Learning methodology Final Selection
### USE Random Forest Model to perform future bike usage prediction

# Creating a date range for the year 2020
dates_forecast = pd.date_range(start='2020-03-27', end='2023-12-30', freq='H')
df_forecast = pd.DataFrame(dates_forecast, columns=['TIME'])

# Extracting year, month, day, day of week, and hour from the date range
df_forecast['YEAR'] = df_forecast['TIME'].dt.year
df_forecast['MONTH'] = df_forecast['TIME'].dt.month
df_forecast['DAY'] = df_forecast['TIME'].dt.day
df_forecast['WEEKDAY'] = df_forecast['TIME'].dt.weekday
df_forecast['HOUR'] = df_forecast['TIME'].dt.hour

# Preparing the feature set for prediction
X_features = df_forecast[['YEAR', 'MONTH', 'DAY', 'WEEKDAY', 'HOUR']]

# Predicting bike usage for 2020
predicted_bike_usage = rf_model.predict(X_features)
df_forecast['PREDICTED_BIKE_USAGE'] = predicted_bike_usage
df_forecast['PREDICTED_BIKE_USAGE'] = np.floor(df_forecast['PREDICTED_BIKE_USAGE']).astype(int)

# Displaying the first few rows of the prediction
```

```
df_forecast.head()

df_allData_hourly_bike_usage.head()

# Setting the plot size
plt.figure(figsize=(15,6))

# Plotting the actual bike usage in 2019
#plt.plot(df_beforePandemic_hourly_bike_usage['TIME'], df_beforePandemic_hourly_bike_usage['BIKE_USAGE'],
label='Actual Bike Usage before Pandemic', alpha=0.7)
plt.plot(df_allData_hourly_bike_usage['TIME'], df_allData_hourly_bike_usage['BIKE_USAGE'], label='Actual Bike
Usage (2018 - 2023)', alpha=0.7,color='red')

# Plotting the predicted bike usage in 2020
plt.plot(df_forecast['TIME'], df_forecast['PREDICTED_BIKE_USAGE'], label='Predicted Bike Usage (2020 - 2023)',
alpha=0.6, color='green')

# Adding plot title and labels
plt.title('Bike Usage: Actual (2018 - 2023) V.S. Predicted (2020 - 2023)')
plt.xlabel('Time')
plt.ylabel('Bike Usage')

# Adding legend
plt.legend()

# Improving layout
plt.tight_layout()

# Displaying the plot
plt.show()

##
    ##Select few days to check the accuracy of the model
##

# Convert 'TIME' columns to datetime for proper filtering
df_allData_hourly_bike_usage['TIME'] = pd.to_datetime(df_allData_hourly_bike_usage['TIME'])
df_forecast['TIME'] = pd.to_datetime(df_forecast['TIME'])

# Define the start and end date for the period
start_date = pd.Timestamp('2020-03-27')
end_date = pd.Timestamp('2020-04-03')

# Filter the dataframes for the specified dates
filtered_allData = df_allData_hourly_bike_usage[(df_allData_hourly_bike_usage['TIME'] >= start_date) &
(df_allData_hourly_bike_usage['TIME'] <= end_date)]
filtered_predict = df_forecast[(df_forecast['TIME'] >= start_date) & (df_forecast['TIME'] <= end_date)]

# Setting the plot size
```

```
plt.figure(figsize=(15, 6))

# Plotting the actual bike usage
plt.plot(filtered_allData['TIME'], filtered_allData['BIKE_USAGE'], label='Actual Bike Usage (Mar 27 - Apr 3, 2020)', alpha=0.7, color='red')

# Plotting the predicted bike usage
plt.plot(filtered_predict['TIME'], filtered_predict['PREDICTED_BIKE_USAGE'], label='Predicted Bike Usage (Mar 27 - Apr 3, 2020)', alpha=0.6, color='green')

# Adding plot title and labels
plt.title('Bike Usage: Actual vs Predicted (Mar 27 - Apr 3, 2020)')
plt.xlabel('Time')
plt.ylabel('Bike Usage')

# Adding legend
plt.legend()

# Improving layout
plt.tight_layout()

# Displaying the plot
plt.show()

## Evaluation
## Predict bike usage V.S. Real bike usage data statistic analyse

# Combine predict dataset with origin actual dataset
df_org_twoPeriods = df_allData_hourly_bike_usage[df_allData_hourly_bike_usage['TIME'] >= pd.to_datetime('2020-03-27')]
df_org_twoPeriods_combined = pd.merge(df_org_twoPeriods, df_forecast, left_on=['YEAR', 'MONTH', 'DAY', 'HOUR', 'WEEKDAY','TIME'],
                          right_on=['YEAR', 'MONTH', 'DAY', 'HOUR', 'WEEKDAY','TIME'], how='left')
# Keep only the necessary columns from the merged dataframe
df_org_twoPeriods_combined = df_org_twoPeriods_combined[['TIME','YEAR', 'MONTH', 'DAY', 'HOUR', 'WEEKDAY', 'BIKE_USAGE','PREDICTED_BIKE_USAGE' ]]

df_org_twoPeriods_combined.head()

# Convert 'TIME' column to datetime format for easier processing

# Define the pandemic and post-pandemic periods
pandemic_start = pd.to_datetime("2020-03-27")
pandemic_end = pd.to_datetime("2021-10-21")
post_pandemic_start = pd.to_datetime("2021-10-22")
post_pandemic_end = pd.to_datetime("2023-12-30")

# Splitting the dataset into two periods: Pandemic and Post-Pandemic
pandemic_df = df_org_twoPeriods_combined[(df_org_twoPeriods_combined['TIME'] >= pandemic_start) &
```

```
(df_org_twoPeriods_combined['TIME'] <= pandemic_end)]
post_pandemic_df = df_org_twoPeriods_combined[(df_org_twoPeriods_combined['TIME'] >= post_pandemic_start)
& (df_org_twoPeriods_combined['TIME'] <= post_pandemic_end)]

# Statistical analysis for both periods
pandemic_stats = pandemic_df[['BIKE_USAGE', 'PREDICTED_BIKE_USAGE']].describe()
post_pandemic_stats = post_pandemic_df[['BIKE_USAGE', 'PREDICTED_BIKE_USAGE']].describe()

# Creating a new DataFrame to store the results
stats_df = pd.DataFrame()

# Adding Pandemic period statistics
stats_df['Pandemic_Hourly_Usage'] = pandemic_stats['BIKE_USAGE']
stats_df['Pandemic_Predicted_Hourly_Usage'] = pandemic_stats['PREDICTED_BIKE_USAGE']

# Adding Post-Pandemic period statistics
stats_df['Post_Pandemic_Hourly_Usage'] = post_pandemic_stats['BIKE_USAGE']
stats_df['Post_Pandemic_Predicted_Hourly_Usage'] = post_pandemic_stats['PREDICTED_BIKE_USAGE']

# Removing 'count' row and rounding other values to two decimal places
stats_df = stats_df.drop('count').map(lambda x: round(x, 2))

stats_df


# Plotting histograms with more distinct colors for better visual differentiation
plt.figure(figsize=(15, 6))

# Pandemic Period Histogram with distinct colors
plt.subplot(1, 2, 1)
plt.hist(pandemic_df['BIKE_USAGE'], bins=30, alpha=0.7, color='blue', label='Actual Bike Usage')
plt.hist(pandemic_df['PREDICTED_BIKE_USAGE'], bins=30, alpha=0.7, color='red', label='Predicted Bike Usage')
plt.title('Pandemic Period: Actual vs Predicted Bike Usage')
plt.xlabel('Bike Usage (per hour)')
plt.ylabel('Frequency')
plt.legend()

# Post-Pandemic Period Histogram with distinct colors
plt.subplot(1, 2, 2)
plt.hist(post_pandemic_df['BIKE_USAGE'], bins=30, alpha=0.7, color='blue', label='Actual Bike Usage')
plt.hist(post_pandemic_df['PREDICTED_BIKE_USAGE'], bins=30, alpha=0.7, color='red', label='Predicted Bike
Usage')
plt.title('Post-Pandemic Period: Actual vs Predicted Bike Usage')
plt.xlabel('Bike Usage (per hour)')
plt.ylabel('Frequency')
plt.legend()

plt.tight_layout()
plt.show()
```