Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

TRINITY COLLEGE DUBLIN
School of Computer Science and Statistics
Final Assignment CS7CS4 Machine Learning

Nachiketh Janapareddy **23337215**

Part 1

Introduction

In the wake of the COVID-19 pandemic, Dublin's city-bike system, like many urban mobility services, faced unprecedented challenges. The pandemic's onset and subsequent waves led to a seismic shift in transportation behaviors, with lockdowns and a surge in remote work significantly diminishing commuter reliance on public transit systems. This study, therefore, focuses on key features such as geographical data, temporal patterns, and weather conditions, chosen for their potential to unveil the altered rhythms of city-bike usage during and after the pandemic. By integrating these variables, our analysis aims to shed light on the pandemic's tangible impacts and provide a predictive outlook for the future. These insights will serve as a cornerstone for policymakers and urban planners as they recalibrate the bike-sharing network to align with the evolving patterns of city life in the new normal.

Data Pre-processing and Feature engineering

In the Data Preparation phase, we loaded and merged Dublinbikes and weather datasets, aligning them chronologically and standardizing the format and headers. Through feature engineering, we extracted time-related elements like time of day and week from 'TIME', revealing usage patterns. The 'bike_usage_daily' metric sums the absolute changes in bike numbers per station daily, indicating total check-ins and outs. This summation represents the total number of bikes that were either checked out or returned to a station over the course of a single day, thus serving as a proxy for daily bike usage at each station.

The histograms presented offer a chronological visualization of bike usage frequencies in Dublin from before and during the pandemic to the post-pandemic period, spanning 2018 to 2023. In the pre-pandemic years of 2018 and 2019, we observe bell-shaped distributions, typical for stable and regular usage patterns. The year 2020 shows a significant decline in usage, reflecting the immediate impact of lockdowns and restrictions. By 2021, there's a noticeable recovery in bike usage, hinting at adaptation to the ongoing pandemic or eased restrictions. In 2022, the distribution begins to normalize, suggesting a return to pre-pandemic mobility levels.

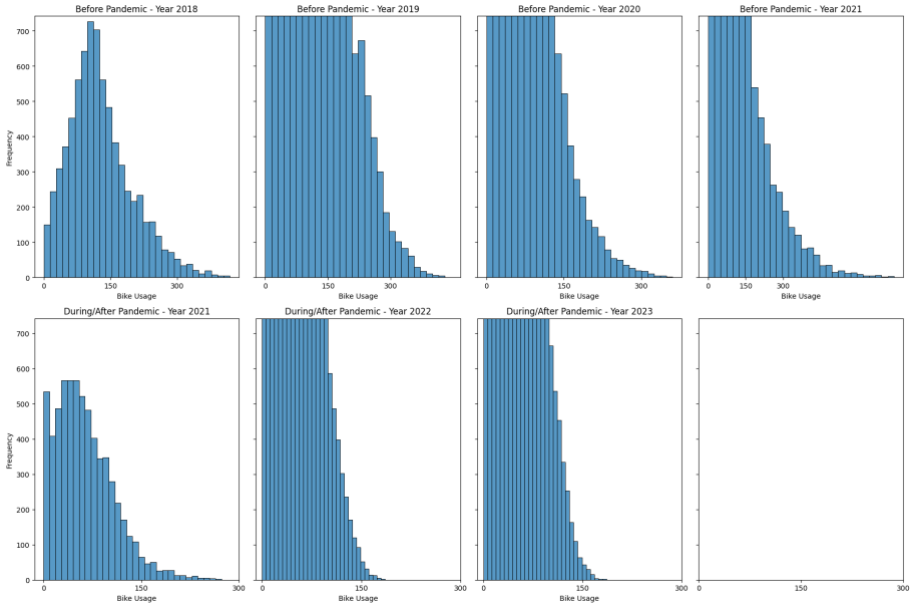
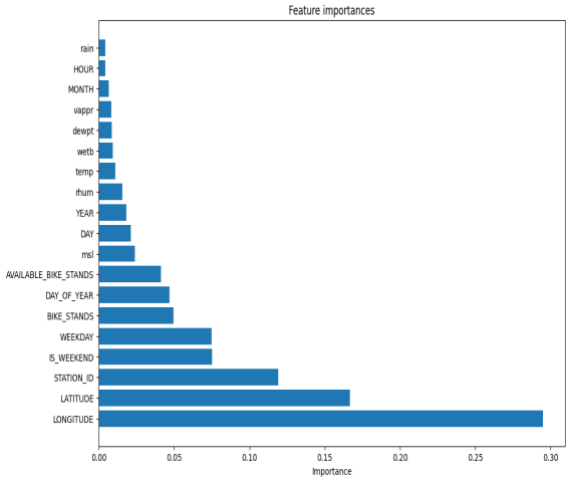
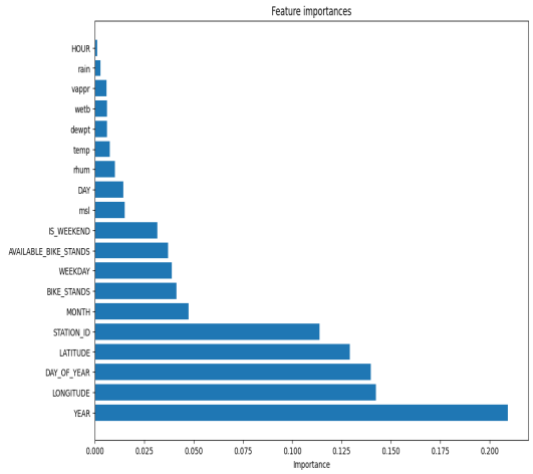


Fig 1 – Visualisation of Actual data of bike usage



Task 1



Task 2

Fig 2 – Feature importance visualisation

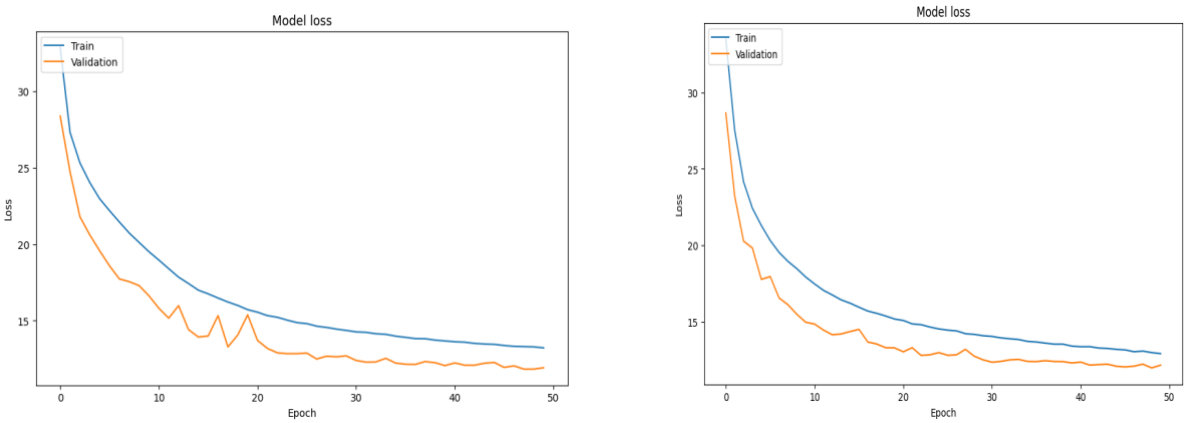
The RandomForestRegressor uses multiple decision trees to determine feature importance, identifying how each attribute affects predictive variability. It measures each feature's impact by observing the mix-up in prediction error when the feature values are permuted, thereby reflecting its importance. Key features like 'LONGITUDE', 'LATITUDE', and 'YEAR' highlight significant factors like location and evolving trends, especially under the pandemic's influence. Other attributes like 'DAY_OF_YEAR', 'STATION_ID', and 'BIKE_STANDS' are consistently significant, indicating the impact of time, location, and infrastructure on bike usage. A noted decrease in the importance of 'HOUR' and weather elements suggests a shift in user behavior post-pandemic. These insights are critical for urban planning and policy-making, helping adapt to changing bike usage patterns.

To prepare for modeling, we selected key features based on their importance to city-bike usage, including station characteristics, geographical coordinates, temporal factors, and weather conditions. These features were then normalized using MinMaxScaler to ensure uniform scaling and fed into an LSTM model, necessitating a reshaping of the data into a 3D format appropriate for time-series forecasting. This process of careful feature selection and normalization is crucial to developing a robust model that accurately captures the dynamics of bike-sharing usage.

Model

Long Short-Term Memory (LSTM) networks excel in time-series analysis due to their proficiency in recognizing and retaining patterns over lengthy intervals, essential for deciphering the variable nature of bike-sharing usage amidst the pandemic's ups and downs. The layered architecture of LSTMs, particularly when bidirectional, enhances their capacity to interpret both past and future contexts, a feature less sophisticated algorithms lack. This capability proves vital in forecasting where temporal sequences and patterns are complex and intertwined, as observed in the shifts in transportation during COVID-19.

Our chosen LSTM model benefits from a bidirectional structure and batch normalization, fine-tuned to distill the intricate temporal correlations within our dataset. Through hyperparameter tuning—adjusting the model's depth and dropout regularization—we've tailored the network to balance between memorization and generalization, avoiding overfitting while ensuring predictive robustness. The model's compilation with a mean absolute error loss function and mean squared error metric is strategic, prioritizing a balance between forecast precision and the mitigation of potential outliers' impact, which is particularly pertinent given the unpredictable swings in bike usage during various phases of the pandemic.



Task 1 Task 2

Fig 3 – Visualisation for Loss for both Tasks

For Task 1 and Task 2 evaluating city-bike usage during and after the pandemic, training and validation losses converge, indicating a good fit without overfitting. Task 1 shows an MAE of 11.92 and MSE of 321.38, suggesting accurate predictions during the pandemic. Task 2 has a slightly higher MAE and MSE (12.19 and 333.60), implying increased post-pandemic prediction error, possibly due to evolving usage patterns. These results can guide Dublin City Council in optimizing bike-sharing for the changing urban landscape.

Fig 4 – Task 1 histogram visualisation with y axis only till 700 for detailed view

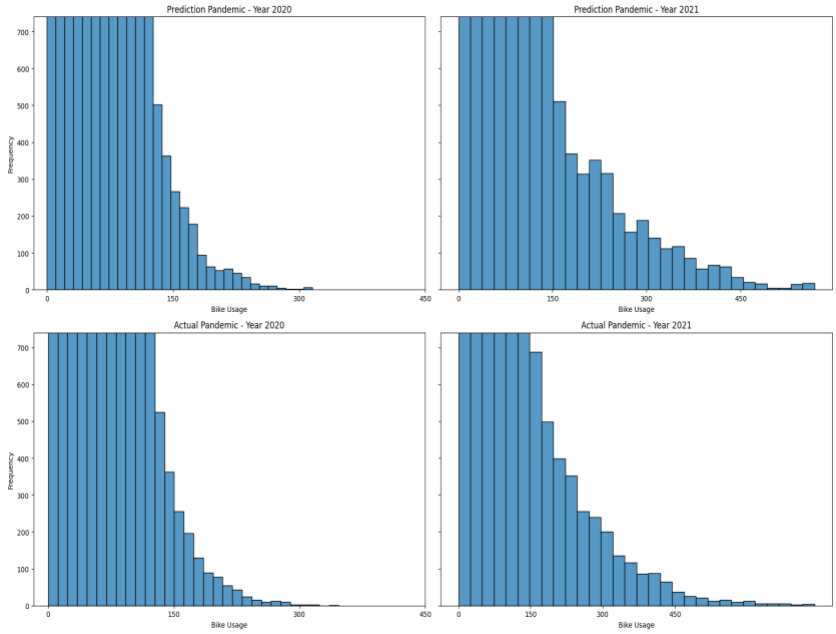
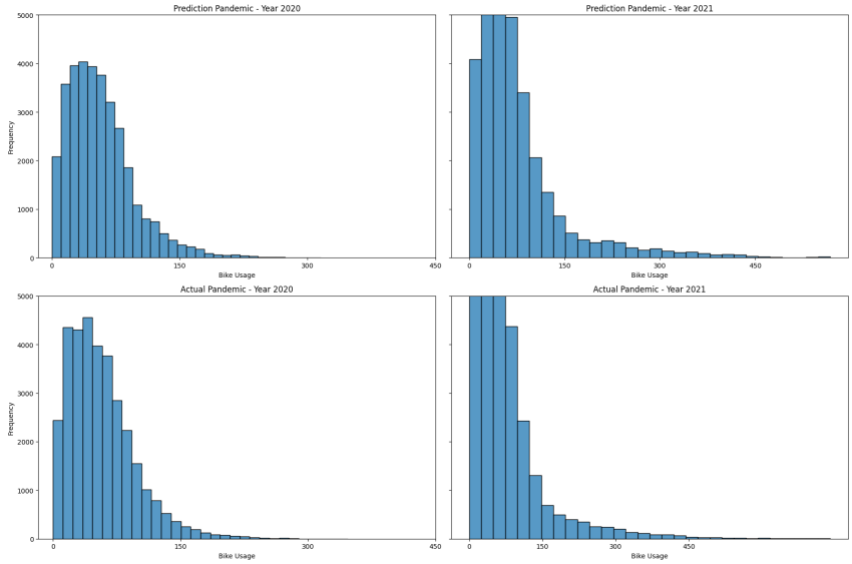


Fig 5 – Task 1 histogram visualisation with y axis only till 5000 for global view



Task 1

The histograms provided illustrate the comparison between the predicted and actual bike usage during the pandemic for the years 2020 and 2021. For both years, the models' predictions capture the overall distribution and trends of bike usage. In 2020, the match between predicted and actual usage is closely aligned, indicating the model's effectiveness in capturing the impact of COVID-19 on bike-sharing patterns during the initial phase of the pandemic. The 2021 histograms show a continuation of this trend, with the model predictions reflecting actual usage, though with some discrepancies that may represent the changing dynamics as the city adapts to the evolving situation. These visual comparisons are essential for assessing the accuracy of the predictive models and understanding the pandemic's impact on bike-sharing services, providing valuable insights for strategic planning and system optimization in response to such unprecedented events.

Fig 6– Task 1 histogram visualisation with y axis only till 700 for detailed view

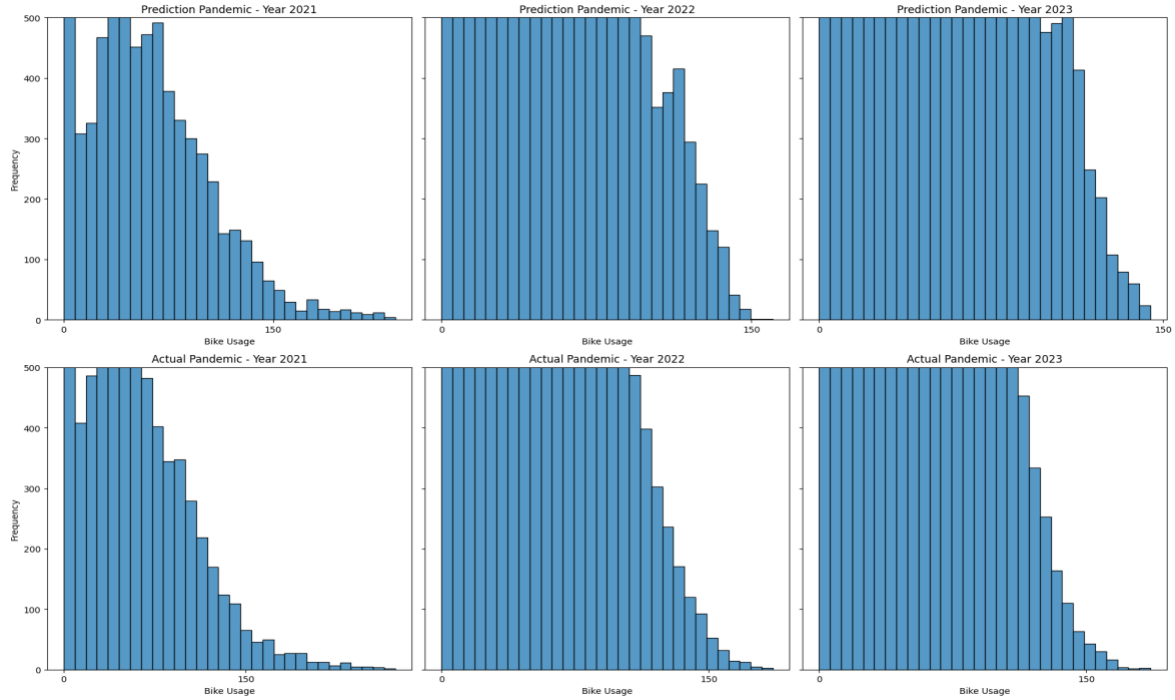
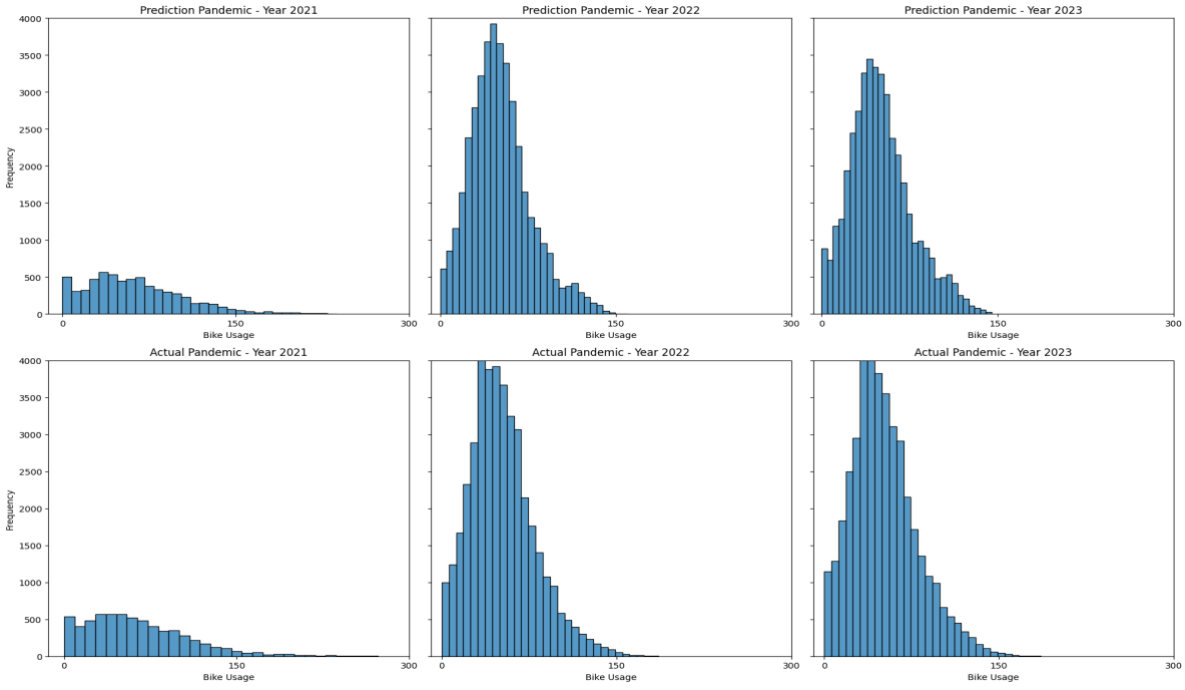


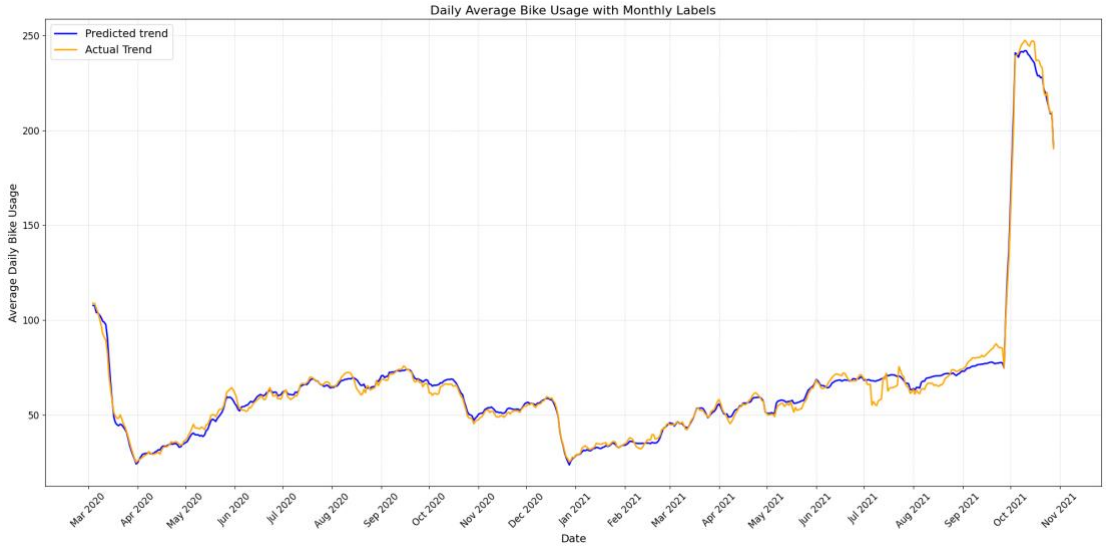
Fig 7 – Task 2 histogram visualisation with y axis only till 5000 for global view



Task 2

The histograms for Task 2 provide a visual comparison between the model's predicted bike usage and the actual data for the years 2021, 2022, and 2023, offering insights into the post-pandemic usage patterns. The model's predictions generally reflect the shape and spread of the actual usage distributions, capturing key characteristics of the dataset. While there is a good degree of alignment between predicted and actual frequencies, some variations are evident, reflecting the complexity of modeling human behavior in a post-pandemic environment where patterns may have shifted due to changes in work habits, public health measures, or personal preferences. These comparisons are pivotal in validating the model's performance and guiding further refinement for predictive accuracy, enabling informed decision-making for the continued development and management of the city's bike-sharing program.

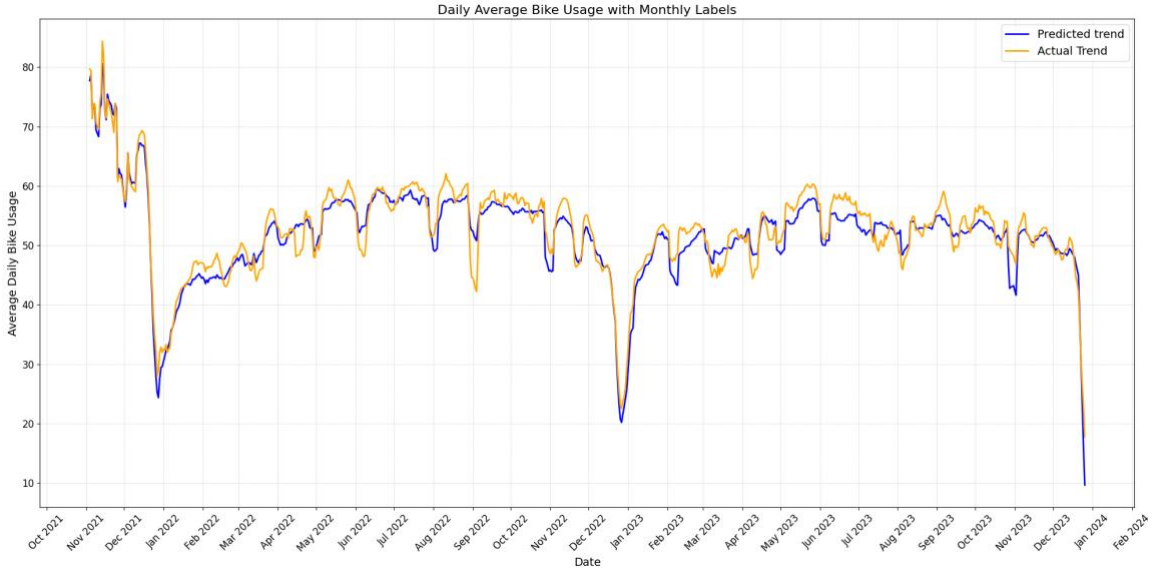
Fig 8 – Task 1 – Visualisation of Trend on monthly timeline



Task 1

The line graph for Task 1 displays the daily average bike usage over time with monthly labels, comparing the predicted trend against the actual trend throughout the year. The predicted trend closely follows the actual usage pattern, with both lines showing a significant dip and recovery, likely corresponding to lockdown and reopening phases during the pandemic. The congruence of the two lines demonstrates the model's capability to approximate real-world bike usage despite the volatile conditions brought about by the pandemic. This visual comparison not only validates the model's performance but also encapsulates the dramatic shifts in public bike-sharing behavior, reflecting the broader impact of COVID-19 on urban mobility.

Fig 9– Task 2 – Visualisation of Trend on monthly timeline



Task 2

The line graph for Task 2 shows the daily average bike usage with monthly labels, comparing the predicted trend with the actual trend post-pandemic. Both the predicted and actual trends closely track each other, indicating the model's effectiveness at capturing the nuances of bike-sharing usage as the city moves forward from the pandemic. Occasional discrepancies between predicted and actual usage may point to external variables or shifts in user behavior not fully encapsulated by the model. This congruence is crucial for stakeholders, as it highlights the model's utility in forecasting and planning in a post-pandemic environment, where understanding the recovery and normalization of bike usage patterns is essential for urban mobility planning.

Data Anomalies or Outliers

Task 1 Insights (During Pandemic):

Identified Anomalies/Outliers

- Sharp Declines in Usage: Observed sharp declines in bike usage during initial lockdown periods, significantly deviating from the predicted trends.
- Unusual Peaks: Noted occasional peaks possibly related to relaxation of restrictions, public events, or temporary shifts in transportation policy.
- Seasonal Effects: Identified a more pronounced decrease during December, likely due to the holiday season and compounded by pandemic restrictions.

Handling Strategies

- Model Robustness: Ensured the LSTM model was robust enough to capture general trends and not overfit to these anomalies.
- Contextual Analysis: Contextualized each anomaly by looking at corresponding dates and external events (e.g., lockdowns, holidays) to understand their impact.
- Data Preprocessing: Applied preprocessing techniques to mitigate the impact of outliers on model training, such as smoothing or anomaly detection methods.

Task 2 Insights (Post-Pandemic):

Identified Anomalies/Outliers

- Gradual Recovery Inconsistencies: Noted inconsistencies in the rate of recovery in bike usage, with some days or periods showing unexpected dips or surges.
- Persistent Seasonal Decline: Continued to see a decrease in December, suggesting that the seasonal impact on bike usage persists post-pandemic.
- Variability in Peak Usage Days: Observed variability in the expected high-usage days, indicating that the recovery might not be as uniform as anticipated.

Handling Strategies

- Adaptive Modeling: Adjusted the model to better adapt to the post-pandemic variability and to be more responsive to changes in user behavior.
- Continuous Monitoring: Recommended continuous monitoring of bike usage trends to quickly identify and analyze any new anomalies that may arise as the city adjusts to post-pandemic norms.
- Comparative Analysis: Conducted a comparative analysis between predicted and actual trends to better understand the nature and impact of these outliers on overall forecasting.

Insights

Task 1 Insights (During Pandemic):

Predicted Behavior (Without Pandemic)

- Peak Usage Expectation: Predictions indicated a continuation of regular high-usage patterns, not accounting for unforeseen disruptions.
- Continuous Growth Trend: Expected a steady or increasing trend based on historical data, with no sudden drops.
- Uniform Daily Distribution: Predicted a more uniform distribution of usage across days, typical of pre-pandemic patterns.

Actual Behavior (With Pandemic)

- Unprecedented Declines: Observed significant drops in usage, especially on historically peak days, due to lockdowns and restrictions.
- Non-Uniform Daily Distribution: Saw erratic and significantly reduced bike usage, reflecting the immediate impact of pandemic measures.
- Base-Level Usage Persistence: Noted that despite overall reduced usage, there remained some consistent level of activity, likely for essential travel.

Task 2 Insights (Post-Pandemic):

Predicted Behavior (Post-Pandemic Expectation)

- Optimistic Rebound: Predicted a quick rebound to near pre-pandemic usage levels, assuming a rapid societal recovery.
- Steady Recovery Trend: Anticipated a consistent and stable increase in bike usage over time, returning to normal patterns.
- Predicted Return to High-Usage Days: Expected certain days to regain their status as peak usage days, similar to pre-pandemic times.

Actual Behavior (Post-Pandemic Reality)

- Cautious Recovery: Actual data showed a more gradual increase in bike usage, suggesting a more cautious or staggered return to pre-pandemic mobility patterns.
- Stabilization with Variability: Observed a stabilization in bike usage with considerable variability, indicating ongoing adjustments or changes in public behavior and preferences.
- Mismatch on Expected High-Usage Days: Actual data often showed less pronounced increases on anticipated high-usage days, pointing to a slower or different kind of recovery than predicted.

Conclusion

In conclusion, this study analyzed the impact of COVID-19 on Dublin's city-bike usage with an LSTM model, forecasting usage patterns during and after the pandemic. It revealed significant changes in urban mobility, including reduced bike usage and cautious post-pandemic recovery, against a backdrop of consistent seasonal patterns. Using RandomForestRegressor for feature importance analysis, it identified geographical and temporal factors as key to understanding bike usage changes. The study also noted the unpredictable nature of societal disruptions and emphasized the need for adaptable models. It addressed anomalies and offered insights for urban planning and policy, advocating for flexible, data-driven approaches to enhance city resilience and adaptability post-pandemic.

Part 2

i. A ROC (Receiver Operating Characteristic) curve is a plot that shows the performance of a binary classifier as its discrimination threshold is varied. It plots the true positive rate (TPR) against the false positive rate (FPR) for various threshold values, creating a curve. The area under this curve (AUC) is a performance measure; the higher the AUC, the better the classifier. ROC curves are particularly useful over simple accuracy in cases of class imbalance or when different trade-offs between true and false positive rates are desired. They allow for comparison between different classifiers and provide a comprehensive view of performance across all thresholds.

ii. Sure, here are two situations condensed with distinct sections for each:

1. Non-linear Relationships:
    a. Situation: The actual relationship between variables is non-linear, but linear regression is applied.
    b. Reasoning: Linear regression assumes a straight-line relationship and can't capture curves or complex patterns inherent in the data.
    c. Solution: Use non-linear models like polynomial regression or other machine learning methods that capture non-linearity.

2. Presence of Outliers:
    d. Situation: Data contains extreme values that are far from other observations.
    e. Reasoning: Linear regression is sensitive to outliers, which can disproportionately pull the regression line and result in inaccurate predictions.
    f. Solution: Apply robust regression techniques, remove or adjust outliers, or use regularization methods like Ridge or Lasso regression.

iii. SVM Kernel:

Meaning: Transforms data into a higher-dimensional space to make it linearly separable.
Usefulness: Vital for handling non-linear data by allowing linear separation in higher-dimensional space.
Kernels For SVM:

- Linear Kernel: Suitable for linearly separable data.
- Polynomial Kernel: Transforms data into a specified degree of polynomial, useful for non-linear data.
- Radial Basis Function (RBF) / Gaussian Kernel: Useful for non-linear problems, where the decision boundary is not linear.
- Sigmoid Kernel: Mirrors the sigmoid function, often used in neural networks.

CNN Kernel:
Meaning: A small matrix used in convolutional layers to extract features from input data.
Usefulness: Essential for capturing spatial hierarchies and patterns in visual data, aiding in tasks like image classification.
Kernels For CNN:

- Small Kernels (e.g., 3x3): Typically used to capture immediate, local features within the data.
- Large Kernels (e.g., 7x7): Capture more global information from the input data, less common due to higher computational cost.
- Sobel Filters: Specifically designed for edge detection tasks within images.
- Custom Filters: Can be designed or learned for specific features or tasks.

Discussion:
SVM Kernels: Enable complex, non-linear separations in transformed feature spaces.
CNN Kernels: Extract local features and reduce dimensions, adjusting through training to highlight essential patterns for recognition tasks.

iv. Idea Behind Resampling in k-Fold Cross-Validation:

Purpose: Resampling in k-fold cross-validation improves model generalization by using each data point for both training and validation across folds, reducing bias and ensuring the performance metric reflects the model's ability to work with new, unseen data.
Example: In 5-fold cross-validation with 100 data points, the dataset is divided into 5 parts. Each part sequentially serves as the test set with the remaining parts used for training, iterating 5 times. The model's performance is averaged across all tests.

When to Use k-Fold Cross-Validation:
Appropriate: It's suitable when data is limited, requiring robust performance estimation, or when the dataset is balanced. It ensures every data point is used, maximizing data efficiency.
Not Appropriate: It's less suitable for very large datasets (due to computational cost), time series data (due to breaking of sequence), or highly imbalanced datasets (as folds might not represent data well). In such cases, alternative methods or adjustments are needed.

Appendix

Task 1-

```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import seaborn as sns

# Define the directory where the CSV files are located
dataset_directory = '/Users/nachiketh/Library/CloudStorage/OneDrive-
TrinityCollegeDublin/Machine_Learning/final project/dataset'  # Update this to your actual dataset
directory

# Load the data
csv_files = [f for f in os.listdir(dataset_directory) if f.endswith('.csv')]
csv_files.sort()

all_data = pd.DataFrame()

for file in csv_files:
    file_path = os.path.join(dataset_directory, file)
    temp_data = pd.read_csv(file_path)
    temp_data['TIME'] = pd.to_datetime(temp_data['TIME'])
    temp_data.columns = [col.replace(" ", "_").upper() for col in temp_data.columns]
    all_data = pd.concat([all_data, temp_data], ignore_index=True)

# Feature Engineering

import pandas as pd
import os
import matplotlib.pyplot as plt
import seaborn as sns


# Function to plot histograms
def plot_histograms(data, years, period_title, ax, max_frequency):
    for i, year in enumerate(years):
        year_data = data[data['TIME'].dt.year == year]['BIKE_USAGE']
        sns.histplot(year_data, bins=30, ax=ax[i], kde=False)
        ax[i].set_title(f'{period_title} - Year {year}')
        ax[i].set_xlabel('Bike Usage')
        ax[i].set_ylabel('Frequency')
        ax[i].set_ylim(0, max_frequency)  # Set the same y-axis limit

dataset_directory = '/Users/nachiketh/Library/CloudStorage/OneDrive-
TrinityCollegeDublin/Machine_Learning/final project/dataset/weather'  # Update this to your actual
dataset directory
csv_files = [f for f in os.listdir(dataset_directory) if f.endswith('.csv')]
csv_files.sort()
```

```
weather_data = pd.DataFrame()

for file in csv_files:
    file_path = os.path.join(dataset_directory, file)
    temp_data = pd.read_csv(file_path)
    weather_data = pd.concat([weather_data, temp_data], ignore_index=True)
    weather_data['date'] = pd.to_datetime(weather_data['date'])


weather_data.head()

all_data = all_data.sort_values('TIME')
weather_data = weather_data.sort_values('date')

# Perform the asof merge
# This assumes that df1 is the left DataFrame and df2 is the right DataFrame
merged_df = pd.merge_asof(all_data, weather_data, left_on='TIME', right_on='date', direction='nearest')

merged_df['DAY'] = merged_df['TIME'].dt.day
merged_df['HOUR'] = merged_df['TIME'].dt.hour
merged_df['WEEKDAY'] = merged_df['TIME'].dt.weekday
merged_df['MONTH'] = merged_df['TIME'].dt.month
merged_df['YEAR'] = merged_df['TIME'].dt.year
merged_df['DAY_OF_YEAR'] = merged_df['TIME'].dt.dayofyear
merged_df['WEEK_OF_YEAR'] = merged_df['TIME'].dt.isocalendar().week
merged_df['IS_WEEKEND'] = (merged_df['WEEKDAY'] >= 5).astype(int)

# Calculate DELTA_BIKES before aggregation
merged_df['DATE'] = merged_df['TIME'].dt.date

# Calculate DELTA_BIKES before aggregation
merged_df.sort_values(by=['STATION_ID', 'TIME'], inplace=True)
merged_df['DELTA_BIKES'] = merged_df.groupby('STATION_ID')['AVAILABLE_BIKES'].diff().abs()

# Now, aggregate DELTA_BIKES to get daily BIKE_USAGE for each station
bike_usage_daily = merged_df.groupby(['STATION_ID',
'DATE'])['DELTA_BIKES'].sum().reset_index(name='BIKE_USAGE')

# Merge the aggregated data back with the merged_df to add the engineered features
bike_usage_daily = bike_usage_daily.merge(
    merged_df.drop_duplicates(subset=['STATION_ID', 'DATE']),
    left_on=['STATION_ID', 'DATE'],
    right_on=['STATION_ID', 'DATE'],
    how='left'
)

# Drop columns that are not needed after the merge
columns_to_drop = ['DATE', 'AVAILABLE_BIKES', 'DELTA_BIKES']
bike_usage_daily.drop(columns=columns_to_drop, inplace=True)

# Ensure 'TIME' is in datetime format after merging
bike_usage_daily['TIME'] = pd.to_datetime(bike_usage_daily['TIME'])
```

```
# Plotting part
# Separate the data into pre-pandemic and pandemic periods
# Before pandemic: Before March 2020
daily_before_pandemic = bike_usage_daily[(bike_usage_daily['YEAR'] < 2020) |
                          ((bike_usage_daily['YEAR'] == 2020) & (bike_usage_daily['MONTH'] < 3))]

# During pandemic: March 2020 to December 2021
daily_during_after_pandemic = bike_usage_daily[((bike_usage_daily['YEAR'] == 2020) &
(bike_usage_daily['MONTH'] >= 3)) |
                          (bike_usage_daily['YEAR'] == 2021) & (bike_usage_daily['MONTH'] < 11)]


# Get unique years and sort them
years_before = daily_before_pandemic['YEAR'].unique()
years_during_after = daily_during_after_pandemic['YEAR'].unique()
years_before.sort()
years_during_after.sort()

# Find the maximum bike usage frequency across all years for plotting
max_bike_usage_frequency = max(daily_before_pandemic['BIKE_USAGE'].max(),
                       daily_during_after_pandemic['BIKE_USAGE'].max())

# Set up the matplotlib figure
num_cols = max(len(years_before), len(years_during_after))
fig, axs = plt.subplots(2, num_cols, figsize=(18, 12), sharey=True, squeeze=False)

# Define the plot_histograms function from your original code

# Plot histograms for 'Before Pandemic'
plot_histograms(daily_before_pandemic, years_before, 'Before Pandemic', axs[0, :],
max_bike_usage_frequency)

# Plot histograms for 'During/After Pandemic'
plot_histograms(daily_during_after_pandemic, years_during_after, 'During/After Pandemic', axs[1, :],
max_bike_usage_frequency)

# ... (your existing code for plotting histograms)

# Find the maximum bike usage frequency across all years for plotting
max_bike_usage = bike_usage_daily['BIKE_USAGE'].max()

# Define the x-axis tick intervals you want to use, e.g., every 100 units
x_ticks_interval = range(0, int(max_bike_usage + 1), 150)  # Adjust the step as needed for your data

# Set the x-axis ticks for all subplots
for ax in axs.flatten():
    ax.set_xticks(x_ticks_interval)


# Show the plot
```

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

# Adjust the layout and display the plot
plt.tight_layout()
plt.show()
print(daily_before_pandemic.columns)
numerical_features = daily_before_pandemic.select_dtypes(include=['int64',
'float64','int32']).columns.tolist()
print(numerical_features)
# Assuming 'bike_usage_daily' is your DataFrame

from sklearn.ensemble import RandomForestRegressor  # or RandomForestClassifier, depending on your
task
from sklearn.model_selection import train_test_split

# Assuming bike_usage_daily is your main DataFrame and it's properly preprocessed
# Define your features (X) and target variable (y)
features_without_target = [col for col in numerical_features if col != 'BIKE_USAGE']

X = daily_before_pandemic[features_without_target]
# drop non-numeric or identifier columns
y = daily_before_pandemic['BIKE_USAGE']

# Split 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)

# Initialize and train the model
model = RandomForestRegressor(random_state=42)  # Use RandomForestClassifier if it's a classification
task
model.fit(X_train, y_train)

# Evaluate feature importances
importances = model.feature_importances_
feature_names = X.columns

# Sort the features by importance
sorted_feature_importance = sorted(zip(importances, feature_names), reverse=True)

# Print each feature with its importance
for importance, name in sorted_feature_importance:
    print(f"{name}: {importance}")

# Consider plotting for a better visual representation
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 8))
plt.title("Feature importances")
plt.barh([name for importance, name in sorted_feature_importance], [importance for importance, name in
sorted_feature_importance])
plt.xlabel("Importance")
plt.show()
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import numpy as np

# Define the features to keep
features_to_keep = ['STATION_ID', 'BIKE_STANDS', 'LATITUDE', 'LONGITUDE', 'DAY',
                    'WEEKDAY', 'MONTH', 'YEAR', 'DAY_OF_YEAR', 'IS_WEEKEND',
                     'rain', 'temp', 'wetb', 'dewpt', 'vappr', 'rhum', 'msl']


# Subset the DataFrame to include only the selected features plus the target
selected_features = bike_usage_daily[features_to_keep + ['BIKE_USAGE']]

# Convert selected features to numpy array
values = selected_features.values

# Ensure all data is float
values = values.astype('float32')

# Split the data into training and testing sets
X, y = values[:, :-1], values[:, -1]  # Separating out the features and target variable
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Normalize features with MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
scaler.fit(X_train)  # Fit the scaler using only the training data

# Transform both training and testing data
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Reshape input to be 3D [samples, timesteps, features] for LSTM
X_train_scaled = X_train_scaled.reshape((X_train_scaled.shape[0], 1, X_train_scaled.shape[1]))
X_test_scaled = X_test_scaled.reshape((X_test_scaled.shape[0], 1, X_test_scaled.shape[1]))

import tensorflow as tf
from keras.models import Sequential
from keras.layers import LSTM, Dropout, Dense, Bidirectional, BatchNormalization

def build_complex_model(hp):
    model = Sequential()

    # First LSTM layer with more units and return_sequences=True to stack another LSTM layer
    model.add(Bidirectional(LSTM(
        units=hp.Int('units_1', min_value=64, max_value=512, step=32),
        input_shape=(X_train_scaled.shape[1], X_train_scaled.shape[2]),
        return_sequences=True  # Allows stacking another LSTM layer
    )))
    model.add(Dropout(
        rate=hp.Float('dropout_1', min_value=0.0, max_value=0.5, step=0.1)
    ))
```

```
    # Batch Normalization layer after LSTM layer
    model.add(BatchNormalization())

    # Second LSTM layer with increased complexity
    model.add(Bidirectional(LSTM(
        units=hp.Int('units_2', min_value=64, max_value=512, step=32),
        return_sequences=True  # Allows stacking another LSTM layer
    )))
    model.add(Dropout(
        rate=hp.Float('dropout_2', min_value=0.0, max_value=0.5, step=0.1)
    ))

    # Batch Normalization layer after LSTM layer
    model.add(BatchNormalization())

    # Third LSTM layer, assuming it's the final LSTM layer
    model.add(Bidirectional(LSTM(
        units=hp.Int('units_3', min_value=64, max_value=512, step=32),
        return_sequences=False  # Last LSTM layer should not return sequences
    )))

    # Dense output layer for prediction, with ReLU activation to ensure non-negative predictions
    model.add(Dense(1, activation='relu'))

    # Compile the model
    model.compile(
    optimizer=tf.keras.optimizers.Adam(
        hp.Float('learning_rate', min_value=1e-4, max_value=1e-2, sampling='LOG')
    ),
    loss='mean_absolute_error',  # primary loss function
    metrics=['mse']  # additional metric to monitor
    )


    return model
# Initialize the tuner
from kerastuner.tuners import RandomSearch
from keras.callbacks import EarlyStopping

tuner = RandomSearch(
    build_complex_model,
    objective='val_loss',
    max_trials=8,  # Higher number of trials for more exhaustive search
    executions_per_trial=1,
    directory='my_dir',
    project_name='bike_usage_tuning_without_availablebikes_with_Weather'
)

early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Perform hyperparameter tuning
```

```
tuner.search(
    X_train_scaled, y_train,
    epochs=10,  # Increased number of epochs
    validation_data=(X_test_scaled, y_test),
    callbacks=[early_stopping],
    verbose=2
)


# Retrieve the best model.
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build the model with the best hyperparameters.
model = build_complex_model(best_hps)

# Summary of the best model
# model.summary()

# Fit the best model
history = model.fit(
    X_train_scaled,
    y_train,
    epochs=50,  # Adjust based on when the model performance converges
    validation_data=(X_test_scaled, y_test),
    verbose=2
)

mae = model.evaluate(X_test_scaled, y_test, verbose=0)
print(f'Test MAE: {mae}')
import matplotlib.pyplot as plt

# Plot training & validation loss values
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()



# 1. Select Features
final_selected_features = daily_during_after_pandemic[features_to_keep]

# 2. Convert to numpy array and ensure float type
final_values = final_selected_features.values
final_values = final_values.astype('float32')
# print(final_selected_features.columns)

# 3. Normalize features with the same MinMaxScaler
```

```
final_values_scaled = scaler.transform(final_values)

# 4. Reshape input to be 3D [samples, timesteps, features] for LSTM
final_values_scaled = final_values_scaled.reshape((final_values_scaled.shape[0], 1,
final_values_scaled.shape[1]))


# Make predictions

predictions = model.predict(final_values_scaled)



def plot_histograms_2(data, years, period_title, ax, max_frequency):
   for i, year in enumerate(years):
      year_data = data[data['YEAR'] == year]['BIKE_USAGE']
      sns.histplot(year_data, bins=30, ax=ax[i], kde=False)
      ax[i].set_title(f'{period_title} - Year {year}')
      ax[i].set_xlabel('Bike Usage')
      ax[i].set_ylabel('Frequency')
      ax[i].set_ylim(0, max_frequency)
negatives_mask = y< 0

# Count the number of negative values
num_negatives = np.sum(negatives_mask)

print("Number of negative values: for training ", num_negatives)
print("Number of positive values: for training", len(y)- num_negatives)

negatives_mask = predictions< 0

# Count the number of negative values
num_negatives = np.sum(negatives_mask)

print("Number of negative values: for pred ", num_negatives)
print("Number of positive values for pred:", len(predictions)- num_negatives)

average_bike_usage = daily_during_after_pandemic['BIKE_USAGE'].mean()
print(average_bike_usage)

# Add predictions to the bike_usage_daily DataFrame
final_selected_features['BIKE_USAGE'] = predictions
max_bike_usage_frequency_2 = max(daily_during_after_pandemic['BIKE_USAGE'].max(),
final_selected_features['BIKE_USAGE'].max())
num_cols = len(years_during_after)
fig, axs = plt.subplots(2, num_cols, figsize=(18, 12), sharey=True, squeeze=False)

# Define the plot_histograms function from your original code

# Plot histograms for 'Before Pandemic'
plot_histograms_2(final_selected_features, years_during_after, 'Prediction Pandemic', axs[0, :],
max_bike_usage_frequency_2)
```

```
# Plot histograms for 'During/After Pandemic'
plot_histograms(daily_during_after_pandemic, years_during_after, 'Actual Pandemic', axs[1, :],
max_bike_usage_frequency_2)

x_ticks_interval = range(0, int(599 + 1), 150)  # Adjust the step as needed for your data

# Set the x-axis ticks for all subplots
for ax in axs.flatten():
   ax.set_xticks(x_ticks_interval)
# Adjust the layout and display the plot
plt.tight_layout()
plt.show()
# final_selected_features.columns
fig, axs = plt.subplots(2, num_cols, figsize=(18, 12), sharey=True, squeeze=False)

# Define the plot_histograms function from your original code

# Plot histograms for 'Before Pandemic'
plot_histograms_2(final_selected_features, years_during_after, 'Prediction Pandemic', axs[0, :], 5000)

# Plot histograms for 'During/After Pandemic'
plot_histograms(daily_during_after_pandemic, years_during_after, 'Actual Pandemic', axs[1, :], 5000)
x_ticks_interval = range(0, int(499 + 1), 150)  # Adjust the step as needed for your data

# Set the x-axis ticks for all subplots
for ax in axs.flatten():
   ax.set_xticks(x_ticks_interval)
# Adjust the layout and display the plot
plt.tight_layout()
plt.show()
final_selected_features['DATE'] = pd.to_datetime(final_selected_features['DAY'].astype(str) + '-' +
                                  final_selected_features['MONTH'].astype(str) + '-' +
                                  final_selected_features['YEAR'].astype(str), format='%d-%m-%Y')

daily_during_after_pandemic['DATE'] = pd.to_datetime(daily_during_after_pandemic['DAY'].astype(str) + '-'
+
                                    daily_during_after_pandemic['MONTH'].astype(str) + '-' +
                                    daily_during_after_pandemic['YEAR'].astype(str), format='%d-%m-%Y')

import matplotlib.pyplot as plt
import pandas as pd

daily_avg_final_features = final_selected_features.groupby('DATE')['BIKE_USAGE'].mean().reset_index()
daily_avg_during_after_pandemic =
daily_during_after_pandemic.groupby('DATE')['BIKE_USAGE'].mean().reset_index()

# Assuming the data is already loaded and grouped as `daily_avg_final_features` and
`daily_avg_during_after_pandemic`
# ...

# Plotting
```

```
plt.figure(figsize=(20, 10))  # Increase figure size for better readability

# Smoothing lines with rolling average; window size can be adjusted
rolling_window_size = 7  # for a weekly rolling average
daily_avg_final_features['BIKE_USAGE_SMOOTH'] =
daily_avg_final_features['BIKE_USAGE'].rolling(window=rolling_window_size, center=True).mean()
daily_avg_during_after_pandemic['BIKE_USAGE_SMOOTH'] =
daily_avg_during_after_pandemic['BIKE_USAGE'].rolling(window=rolling_window_size,
center=True).mean()

plt.plot(daily_avg_final_features['DATE'], daily_avg_final_features['BIKE_USAGE_SMOOTH'],
label='Predicted trend', color='blue', linewidth=2)
plt.plot(daily_avg_during_after_pandemic['DATE'],
daily_avg_during_after_pandemic['BIKE_USAGE_SMOOTH'], label='Actual Trend', color='orange',
linewidth=2)

plt.xlabel('Date', fontsize=14)
plt.ylabel('Average Daily Bike Usage', fontsize=14)
plt.title('Daily Average Bike Usage with Monthly Labels', fontsize=16)

plt.gca().xaxis.set_major_locator(plt.matplotlib.dates.MonthLocator())
plt.gca().xaxis.set_major_formatter(plt.matplotlib.dates.DateFormatter('%b %Y'))
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)

plt.legend(fontsize=14)
plt.grid(True, which='both', linestyle='--', linewidth=0.5, alpha=0.7)
plt.tight_layout()
plt.show()
```

Task 2-

```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import seaborn as sns

# Define the directory where the CSV files are located
dataset_directory = '/Users/nachiketh/Library/CloudStorage/OneDrive-
TrinityCollegeDublin/Machine_Learning/final project/dataset'  # Update this to your actual dataset
directory

# Load the data
csv_files = [f for f in os.listdir(dataset_directory) if f.endswith('.csv')]
csv_files.sort()

all_data = pd.DataFrame()

for file in csv_files:
```

```
    file_path = os.path.join(dataset_directory, file)
    temp_data = pd.read_csv(file_path)
    temp_data['TIME'] = pd.to_datetime(temp_data['TIME'])
    temp_data.columns = [col.replace(" ", "_").upper() for col in temp_data.columns]
    all_data = pd.concat([all_data, temp_data], ignore_index=True)

# Feature Engineering

import pandas as pd
import os
import matplotlib.pyplot as plt
import seaborn as sns


# Function to plot histograms
def plot_histograms(data, years, period_title, ax, max_frequency):
    for i, year in enumerate(years):
        year_data = data[data['TIME'].dt.year == year]['BIKE_USAGE']
        sns.histplot(year_data, bins=30, ax=ax[i], kde=False)
        ax[i].set_title(f'{period_title} - Year {year}')
        ax[i].set_xlabel('Bike Usage')
        ax[i].set_ylabel('Frequency')
        ax[i].set_ylim(0, max_frequency)  # Set the same y-axis limit

dataset_directory = '/Users/nachiketh/Library/CloudStorage/OneDrive-
TrinityCollegeDublin/Machine_Learning/final project/dataset/weather'  # Update this to your actual
dataset directory
csv_files = [f for f in os.listdir(dataset_directory) if f.endswith('.csv')]
csv_files.sort()

weather_data = pd.DataFrame()

for file in csv_files:
    file_path = os.path.join(dataset_directory, file)
    temp_data = pd.read_csv(file_path)
    weather_data = pd.concat([weather_data, temp_data], ignore_index=True)
    weather_data['date'] = pd.to_datetime(weather_data['date'])


weather_data.head()

all_data = all_data.sort_values('TIME')
weather_data = weather_data.sort_values('date')

# Perform the asof merge
# This assumes that df1 is the left DataFrame and df2 is the right DataFrame
merged_df = pd.merge_asof(all_data, weather_data, left_on='TIME', right_on='date', direction='nearest')

merged_df['DAY'] = merged_df['TIME'].dt.day
merged_df['HOUR'] = merged_df['TIME'].dt.hour
merged_df['WEEKDAY'] = merged_df['TIME'].dt.weekday
merged_df['MONTH'] = merged_df['TIME'].dt.month
```

```
merged_df['YEAR'] = merged_df['TIME'].dt.year
merged_df['DAY_OF_YEAR'] = merged_df['TIME'].dt.dayofyear
merged_df['WEEK_OF_YEAR'] = merged_df['TIME'].dt.isocalendar().week
merged_df['IS_WEEKEND'] = (merged_df['WEEKDAY'] >= 5).astype(int)

# Calculate DELTA_BIKES before aggregation
merged_df['DATE'] = merged_df['TIME'].dt.date

# Calculate DELTA_BIKES before aggregation
merged_df.sort_values(by=['STATION_ID', 'TIME'], inplace=True)
merged_df['DELTA_BIKES'] = merged_df.groupby('STATION_ID')['AVAILABLE_BIKES'].diff().abs()

# Now, aggregate DELTA_BIKES to get daily BIKE_USAGE for each station
bike_usage_daily = merged_df.groupby(['STATION_ID',
'DATE'])['DELTA_BIKES'].sum().reset_index(name='BIKE_USAGE')

# Merge the aggregated data back with the merged_df to add the engineered features
bike_usage_daily = bike_usage_daily.merge(
    merged_df.drop_duplicates(subset=['STATION_ID', 'DATE']),
    left_on=['STATION_ID', 'DATE'],
    right_on=['STATION_ID', 'DATE'],
    how='left'
)

# Drop columns that are not needed after the merge
columns_to_drop = ['DATE', 'AVAILABLE_BIKES', 'DELTA_BIKES']
bike_usage_daily.drop(columns=columns_to_drop, inplace=True)

# Ensure 'TIME' is in datetime format after merging
bike_usage_daily['TIME'] = pd.to_datetime(bike_usage_daily['TIME'])


# Plotting part
# Separate the data into pre-pandemic and pandemic periods
# Before pandemic: Before March 2020
daily_before_during_pandemic = bike_usage_daily[(bike_usage_daily['YEAR'] == 2021) &
(bike_usage_daily['MONTH'] <= 10) |
                                (bike_usage_daily['YEAR'] < 2022)]
daily_after_pandemic = bike_usage_daily[(bike_usage_daily['YEAR'] >= 2022) |
                                    (bike_usage_daily['YEAR'] == 2021) & (bike_usage_daily['MONTH'] > 10)]


# Get unique years and sort them
years_before = daily_before_during_pandemic['YEAR'].unique()
years_during_after = daily_after_pandemic['YEAR'].unique()
years_before.sort()
years_during_after.sort()

# Find the maximum bike usage frequency across all years for plotting
max_bike_usage_frequency = max(daily_before_during_pandemic['BIKE_USAGE'].max(),
                        daily_after_pandemic['BIKE_USAGE'].max())
```

```
# Set up the matplotlib figure
num_cols = max(len(years_before), len(years_during_after))
fig, axs = plt.subplots(2, num_cols, figsize=(18, 12), sharey=True, squeeze=False)

# Define the plot_histograms function from your original code

# Plot histograms for 'Before Pandemic'
plot_histograms(daily_before_during_pandemic, years_before, 'Before Pandemic', axs[0, :],
max_bike_usage_frequency)

# Plot histograms for 'During/After Pandemic'
plot_histograms(daily_after_pandemic, years_during_after, 'During/After Pandemic', axs[1, :],
max_bike_usage_frequency)

# ... (your existing code for plotting histograms)

# Find the maximum bike usage frequency across all years for plotting
max_bike_usage = bike_usage_daily['BIKE_USAGE'].max()

# Define the x-axis tick intervals you want to use, e.g., every 100 units
x_ticks_interval = range(0, int(max_bike_usage + 1), 150)  # Adjust the step as needed for your data

# Set the x-axis ticks for all subplots
for ax in axs.flatten():
    ax.set_xticks(x_ticks_interval)


# Show the plot
plt.tight_layout()
plt.show()

# Adjust the layout and display the plot
plt.tight_layout()
plt.show()
print(daily_before_during_pandemic.columns)
numerical_features = daily_before_during_pandemic.select_dtypes(include=['int64',
'float64','int32']).columns.tolist()
print(numerical_features)
# Assuming 'bike_usage_daily' is your DataFrame

from sklearn.ensemble import RandomForestRegressor  # or RandomForestClassifier, depending on your
task
from sklearn.model_selection import train_test_split

# Assuming bike_usage_daily is your main DataFrame and it's properly preprocessed
# Define your features (X) and target variable (y)
features_without_target = [col for col in numerical_features if col != 'BIKE_USAGE']

X = daily_before_during_pandemic[features_without_target]
# drop non-numeric or identifier columns
y = daily_before_during_pandemic['BIKE_USAGE']
```

```
# Split 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)

# Initialize and train the model
model = RandomForestRegressor(random_state=42)  # Use RandomForestClassifier if it's a classification
task
model.fit(X_train, y_train)

# Evaluate feature importances
importances = model.feature_importances_
feature_names = X.columns

# Sort the features by importance
sorted_feature_importance = sorted(zip(importances, feature_names), reverse=True)

# Print each feature with its importance
for importance, name in sorted_feature_importance:
    print(f"{name}: {importance}")

# Consider plotting for a better visual representation
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 8))
plt.title("Feature importances")
plt.barh([name for importance, name in sorted_feature_importance], [importance for importance, name in
sorted_feature_importance])
plt.xlabel("Importance")
plt.show()

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import numpy as np

# Define the features to keep
features_to_keep = ['STATION_ID', 'BIKE_STANDS', 'LATITUDE', 'LONGITUDE', 'DAY',
                    'WEEKDAY', 'MONTH', 'YEAR', 'DAY_OF_YEAR', 'IS_WEEKEND',
                     'rain', 'temp', 'wetb', 'dewpt', 'vappr', 'rhum', 'msl']


# Subset the DataFrame to include only the selected features plus the target
selected_features = bike_usage_daily[features_to_keep + ['BIKE_USAGE']]

# Convert selected features to numpy array
values = selected_features.values

# Ensure all data is float
values = values.astype('float32')

# Split the data into training and testing sets
X, y = values[:, :-1], values[:, -1]  # Separating out the features and target variable
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Normalize features with MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
scaler.fit(X_train)  # Fit the scaler using only the training data

# Transform both training and testing data
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Reshape input to be 3D [samples, timesteps, features] for LSTM
X_train_scaled = X_train_scaled.reshape((X_train_scaled.shape[0], 1, X_train_scaled.shape[1]))
X_test_scaled = X_test_scaled.reshape((X_test_scaled.shape[0], 1, X_test_scaled.shape[1]))

import tensorflow as tf
from keras.models import Sequential
from keras.layers import LSTM, Dropout, Dense, Bidirectional, BatchNormalization

def build_complex_model(hp):
    model = Sequential()

    # First LSTM layer with more units and return_sequences=True to stack another LSTM layer
    model.add(Bidirectional(LSTM(
        units=hp.Int('units_1', min_value=64, max_value=512, step=32),
        input_shape=(X_train_scaled.shape[1], X_train_scaled.shape[2]),
        return_sequences=True  # Allows stacking another LSTM layer
    )))
    model.add(Dropout(
        rate=hp.Float('dropout_1', min_value=0.0, max_value=0.5, step=0.1)
    ))

    # Batch Normalization layer after LSTM layer
    model.add(BatchNormalization())

    # Second LSTM layer with increased complexity
    model.add(Bidirectional(LSTM(
        units=hp.Int('units_2', min_value=64, max_value=512, step=32),
        return_sequences=True  # Allows stacking another LSTM layer
    )))
    model.add(Dropout(
        rate=hp.Float('dropout_2', min_value=0.0, max_value=0.5, step=0.1)
    ))

    # Batch Normalization layer after LSTM layer
    model.add(BatchNormalization())

    # Third LSTM layer, assuming it's the final LSTM layer
    model.add(Bidirectional(LSTM(
        units=hp.Int('units_3', min_value=64, max_value=512, step=32),
        return_sequences=False  # Last LSTM layer should not return sequences
    )))

    # Dense output layer for prediction, with ReLU activation to ensure non-negative predictions
    model.add(Dense(1, activation='relu'))
```

```
    # Compile the model
    model.compile(
    optimizer=tf.keras.optimizers.Adam(
        hp.Float('learning_rate', min_value=1e-4, max_value=1e-2, sampling='LOG')
    ),
    loss='mean_absolute_error',  # primary loss function
    metrics=['mse']  # additional metric to monitor
    )


    return model
# Initialize the tuner
from kerastuner.tuners import RandomSearch
from keras.callbacks import EarlyStopping

tuner = RandomSearch(
    build_complex_model,
    objective='val_loss',
    max_trials=8,  # Higher number of trials for more exhaustive search
    executions_per_trial=1,
    directory='my_dir',
    project_name='bike_usage_tuning_post_without_availablebikes_with_Weather'
)

early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Perform hyperparameter tuning
tuner.search(
    X_train_scaled, y_train,
    epochs=10,  # Increased number of epochs
    validation_data=(X_test_scaled, y_test),
    callbacks=[early_stopping],
    verbose=2
)


# Retrieve the best model.
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build the model with the best hyperparameters.
model = build_complex_model(best_hps)

# Summary of the best model
# model.summary()

# Fit the best model
history = model.fit(
    X_train_scaled,
    y_train,
    epochs=50,  # Adjust based on when the model performance converges
    validation_data=(X_test_scaled, y_test),
```

```
    verbose=2
)

mae = model.evaluate(X_test_scaled, y_test, verbose=0)
print(f'Test MAE: {mae}')
import matplotlib.pyplot as plt

# Plot training & validation loss values
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()



# 1. Select Features
final_selected_features = daily_after_pandemic[features_to_keep]

# 2. Convert to numpy array and ensure float type
final_values = final_selected_features.values
final_values = final_values.astype('float32')
# print(final_selected_features.columns)

# 3. Normalize features with the same MinMaxScaler
final_values_scaled = scaler.transform(final_values)

# 4. Reshape input to be 3D [samples, timesteps, features] for LSTM
final_values_scaled = final_values_scaled.reshape((final_values_scaled.shape[0], 1,
final_values_scaled.shape[1]))


# Make predictions

predictions = model.predict(final_values_scaled)



def plot_histograms_2(data, years, period_title, ax, max_frequency):
    for i, year in enumerate(years):
        year_data = data[data['YEAR'] == year]['BIKE_USAGE']
        sns.histplot(year_data, bins=30, ax=ax[i], kde=False)
        ax[i].set_title(f'{period_title} - Year {year}')
        ax[i].set_xlabel('Bike Usage')
        ax[i].set_ylabel('Frequency')
        ax[i].set_ylim(0, max_frequency)
negatives_mask = y< 0

# Count the number of negative values
```

```
num_negatives = np.sum(negatives_mask)

print("Number of negative values: for training ", num_negatives)
print("Number of positive values: for training", len(y)- num_negatives)

negatives_mask = predictions< 0

# Count the number of negative values
num_negatives = np.sum(negatives_mask)

print("Number of negative values: for pred ", num_negatives)
print("Number of positive values for pred:", len(predictions)- num_negatives)

average_bike_usage = daily_after_pandemic['BIKE_USAGE'].mean()
print(average_bike_usage)

# Add predictions to the bike_usage_daily DataFrame
final_selected_features['BIKE_USAGE'] = predictions
max_bike_usage_frequency_2 = max(daily_after_pandemic['BIKE_USAGE'].max(),
final_selected_features['BIKE_USAGE'].max())
num_cols = len(years_during_after)
fig, axs = plt.subplots(2, num_cols, figsize=(18, 12), sharey=True, squeeze=False)

# Define the plot_histograms function from your original code

# Plot histograms for 'Before Pandemic'
plot_histograms_2(final_selected_features, years_during_after, 'Prediction Pandemic', axs[0, :], 500)

# Plot histograms for 'During/After Pandemic'
plot_histograms(daily_after_pandemic, years_during_after, 'Actual Pandemic', axs[1, :], 500)

x_ticks_interval = range(0, int(300), 150)  # Adjust the step as needed for your data

# Set the x-axis ticks for all subplots
for ax in axs.flatten():
    ax.set_xticks(x_ticks_interval)
# Adjust the layout and display the plot
plt.tight_layout()
plt.show()
# final_selected_features.columns
fig, axs = plt.subplots(2, num_cols, figsize=(18, 12), sharey=True, squeeze=False)

# Define the plot_histograms function from your original code

# Plot histograms for 'Before Pandemic'
plot_histograms_2(final_selected_features, years_during_after, 'Prediction Pandemic', axs[0, :], 4000)

# Plot histograms for 'During/After Pandemic'
plot_histograms(daily_after_pandemic, years_during_after, 'Actual Pandemic', axs[1, :], 4000)
x_ticks_interval = range(0, int(350), 150)  # Adjust the step as needed for your data

# Set the x-axis ticks for all subplots
```

```
for ax in axs.flatten():
   ax.set_xticks(x_ticks_interval)
# Adjust the layout and display the plot
plt.tight_layout()
plt.show()
final_selected_features['DATE'] = pd.to_datetime(final_selected_features['DAY'].astype(str) + '-' +
                                  final_selected_features['MONTH'].astype(str) + '-' +
                                  final_selected_features['YEAR'].astype(str), format='%d-%m-%Y')

daily_after_pandemic['DATE'] = pd.to_datetime(daily_after_pandemic['DAY'].astype(str) + '-' +
                                    daily_after_pandemic['MONTH'].astype(str) + '-' +
                                    daily_after_pandemic['YEAR'].astype(str), format='%d-%m-%Y')

import matplotlib.pyplot as plt
import pandas as pd

daily_avg_final_features = final_selected_features.groupby('DATE')['BIKE_USAGE'].mean().reset_index()
daily_avg_during_after_pandemic =
daily_after_pandemic.groupby('DATE')['BIKE_USAGE'].mean().reset_index()

# Assuming the data is already loaded and grouped as `daily_avg_final_features` and
`daily_avg_during_after_pandemic`
# ...

# Plotting
plt.figure(figsize=(20, 10))  # Increase figure size for better readability

# Smoothing lines with rolling average; window size can be adjusted
rolling_window_size = 7  # for a weekly rolling average
daily_avg_final_features['BIKE_USAGE_SMOOTH'] =
daily_avg_final_features['BIKE_USAGE'].rolling(window=rolling_window_size, center=True).mean()
daily_avg_during_after_pandemic['BIKE_USAGE_SMOOTH'] =
daily_avg_during_after_pandemic['BIKE_USAGE'].rolling(window=rolling_window_size,
center=True).mean()

plt.plot(daily_avg_final_features['DATE'], daily_avg_final_features['BIKE_USAGE_SMOOTH'],
label='Predicted trend', color='blue', linewidth=2)
plt.plot(daily_avg_during_after_pandemic['DATE'],
daily_avg_during_after_pandemic['BIKE_USAGE_SMOOTH'], label='Actual Trend', color='orange',
linewidth=2)

plt.xlabel('Date', fontsize=14)
plt.ylabel('Average Daily Bike Usage', fontsize=14)
plt.title('Daily Average Bike Usage with Monthly Labels', fontsize=16)

plt.gca().xaxis.set_major_locator(plt.matplotlib.dates.MonthLocator())
plt.gca().xaxis.set_major_formatter(plt.matplotlib.dates.DateFormatter('%b %Y'))
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)

plt.legend(fontsize=14)
plt.grid(True, which='both', linestyle='--', linewidth=0.5, alpha=0.7)
```

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