Configuration Manual

MSc Research Project
Data Analytics

Aaditya Ravindra Gajendragadkar
StudentID:22158758

School of Computing
National College of Ireland

Supervisor: Furqan Rustam

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Aaditya Ravindra Gajendragadkar
Student ID:	22158758
Programme:	Data Analytics
Year:	2023
Module:	MSc Research Project
Supervisor:	Furqan Rustam
Submission Due Date:	14/12/2023
Project Title:	Configuration Manual
Word Count:	989
Page Count:	15

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Aaditya Ravindra Gajendragadkar
Date:	14th December 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Aaditya Ravindra Gajendragadkar
22158758

1 Introduction

This manual serves as a guide for executing and configuring the implementation code within the scope of the current research project. It offers explicit information about both the machine hardware specifications and the requisite programs for execution. Following the outlined steps will empower users to generate paper summaries utilizing the Models developed during the project.

2 Configuration details

The hardware features an Intel Core i5 processor, 16GB of RAM, 250GB of SSD storage, a dedicated GTX 1650 graphics card, and runs on Windows 11 for smooth performance.

The software system utilized is Jupyter Notebook system is equipped with an Intel Xeon E5-2699 v4 22-core, 44-thread processor with a base clock speed of 2.2 GHz (up to 3.6 GHz with Turbo Boost), complemented by a spacious 250GB storage and an impressive 128GB of RAM, ensuring optimal performance for data-intensive tasks.

3. Software Tools

3.1 Python

Python software was utilized as a programming language in this project. Python stands out as the ideal language for implementation. Its extensive libraries like TensorFlow and PyTorch, sckitlearn coupled with a supportive community, make it a top choice. Python's readability, simplicity, and versatility ease the learning curve, while its adaptability facilitates seamless integration with various technologies. Its industry-wide adoption and active development further solidify Python as the go-to language for cutting-edge machine learning applications detailed in this manual. In addition to this python offers vast visualization libraries. Figure 1 shows

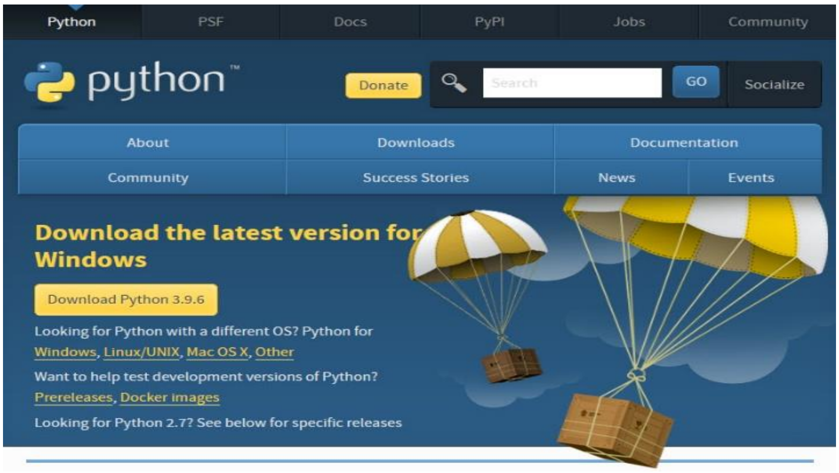


Figure 1: Python image from its official website

3.2 Jupyter Notebook.

Jupyter Notebook was used to code due to its interactive environment, supporting a mix of code, text, and visualizations in a single document. Its versatility extends to multiple programming languages, including Python, R, and Julia. The platform excels in data exploration, and visualization, and fosters reproducibility, making it a preferred choice for collaborative work and educational purposes. Its integration with big data tools and ease of sharing further solidify Jupyter Notebook as a key tool in data science, machine learning, and educational settings. Figure 2 shows jupyter notebook installation from its official website.

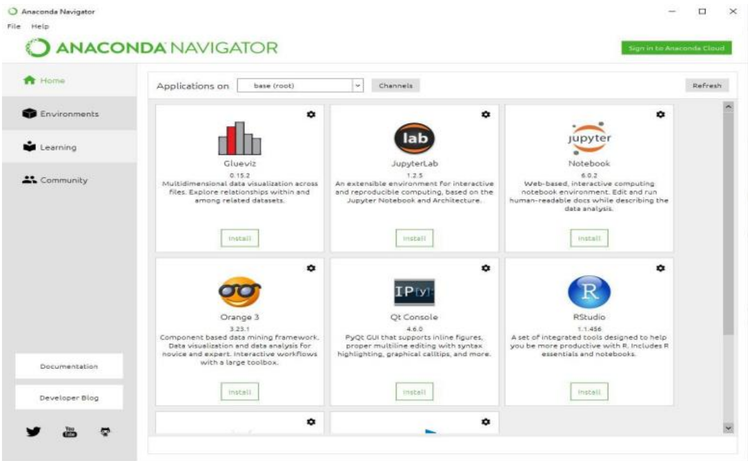


Figure 2 shows the installation of jupyter notebook.

4. Implementation of Project

4.1 Importing libraries.

All the important libraries were downloaded. Figure 3 shows importing all libraries.

```
In [32]:
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import Lasso
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import seaborn as sns

import statsmodels.api as sm
from itertools import product
```

Figure 3 shows importing of all Libraries.

After understanding data frame data was visalised with the help of line charts and scatter plots and understandd the relationship between data.

```
# Read the CSV file
data = pd.read_csv('C:\\Users\\apoor\\Desktop\\NCI\\RP\\traffic_Kaggle.csv')
```

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

# Load your dataset, replace 'your_dataset.csv' with your actual dataset file
# Example: data = pd.read_csv('your_dataset.csv')


# Display basic information about the dataset
print("Dataset Information:")
print(data.info())

# Display summary statistics of numerical columns
print("\nSummary Statistics:")
print(data.describe())
```

```
Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1048575 entries, 0 to 1048574
Data columns (total 5 columns):
 #   Column    Non-Null Count    Dtype
---  ------    --------------    -----
 0   timestep  1048575 non-null  int64
 1   location  1048575 non-null  int64
 2   flow      1048575 non-null  int64
 3   occupy    1048575 non-null  float64
 4   speed     1048575 non-null  float64
dtypes: float64(2), int64(3)
memory usage: 40.0 MB
None

Summary Statistics:
           timestep      location          flow        occupy         speed
count  1.048575e+06  1.048575e+06  1.048575e+06  1.048575e+06  1.048575e+06
mean   3.084544e+03  8.449889e+01  2.270468e+02  6.331729e-02  6.388404e+01
```

Figure 4 shows reading the data frame and understanding the data.

```
# Scatter plot to visualize the relationship between time step and flow for the first 1000 values
plt.figure(figsize=(10, 6))
sns.scatterplot(data=data_first_1000, x='timestep', y='flow')
plt.title('Scatter Plot: Time Step vs. Flow (First 1000 Values)')
plt.xlabel('Time Step')
plt.ylabel('Flow')
plt.show()

# Scatter plot to visualize the relationship between location and flow for the first 1000 values
plt.figure(figsize=(10, 6))
sns.scatterplot(data=data_first_1000, x='location', y='flow')
plt.title('Scatter Plot: Location vs. Flow (First 1000 Values)')
plt.xlabel('Location')
plt.ylabel('Flow')
plt.show()

# Pair plot to visualize relationships between all numeric variables for the first 1000 values
sns.pairplot(data_first_1000, x_vars=['timestep', 'location'], y_vars=['flow'], height=6)
plt.show()

# Heatmap to visualize the correlation between variables for the first 1000 values
correlation_matrix = data_first_1000[['timestep', 'location', 'flow']].corr()
plt.figure(figsize=(8, 6))
```

```
electeu_uata = uata[uata[ iocation ].isin(selecteu_iocations)]

 Line plot for traffic flow over time for each location
lt.figure(figsize=(12, 6))
or location in selected_locations:
   location_data = selected_data[selected_data['location'] == location]
   plt.plot(location_data['timestep'], location_data['flow'], label=f'Location {location}')

lt.title('Traffic Flow Over Time for First Five Locations')
lt.xlabel('Timestep')
lt.ylabel('Flow')
lt.legend()
lt.show()

 Boxplot for traffic flow distribution for each location
lt.figure(figsize=(10, 6))
ns.boxplot(x='location', y='flow', data=selected_data)
lt.title('Traffic Flow Distribution for First Five Locations')
lt.xlabel('Location')
lt.ylabel('Flow')
lt.show()

 Violin plot for traffic flow distribution for each location
lt.figure(figsize=(10, 6))
ns.violinplot(x='location', y='flow', data=selected_data)
lt.title('Traffic Flow Distribution for First Five Locations')
```

Figure 5 shows Data visualisation part of the project.

5. Implementation of models

5.1 ML Models

After visualization of Data implementation of models takes place, Figure 6 shows an implementation of KNN model, decision tree, gradient boost and Random Forest model takes place with train and test splits of varying % ranging from 50 to 90.

.

```
import pandas as pd
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import statsmodels.api as sm

# Assuming you have a DataFrame called 'data' with 'location', 'timestep', and 'flow' columns

# Get unique locations
locations = data['location'].unique()

for location in locations[:5]:
    location_data = data[data['location'] == location]

    # Extract X (independent variables) and y (dependent variable)
    X = location_data[['timestep', 'location']]
    y = location_data['flow']

    # 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 the Decision Tree regression model
    max_depth = 35 # You can adjust the maximum depth of the tree
    model = DecisionTreeRegressor(max_depth=max_depth)

    # Fit the Decision Tree regression model
    model.fit(X, y)
```

```python
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Assuming you have a DataFrame called 'data' with 'location', 'timestep', and 'flow' columns

# Get unique locations
locations = data['location'].unique()

for location in locations[:5]:
    location_data = data[data['location'] == location]

    # Extract X (independent variables) and y (dependent variable)
    X = location_data[['timestep', 'location']]
    y = location_data['flow']

    # 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 the XGBoost Regression model
    xgb_model = XGBRegressor(n_estimators=100, learning_rate=0.1)  # You can adjust parameters accordingly

    # Fit the XGBoost Regression model
    xgb_model.fit(X_train, y_train)

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

```python
locations = data['location'].unique()

for location in locations[:5]:
    location_data = data[data['location'] == location]

    # Extract X (independent variables) and y (dependent variable)
    X = location_data[['timestep', 'location']]
    y = location_data['flow']

    # 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 the Random Forest regression model
    n_estimators = 100  # You can adjust the number of trees in the forest
    model = RandomForestRegressor(n_estimators=n_estimators)

    # Fit the Random Forest regression model
    model.fit(X, y)

    # Make predictions
    y_pred = model.predict(X)

    # Calculate R-squared (R2)
    r2 = r2_score(y, y_pred)
```

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Assuming you have a DataFrame called 'data' with 'location', 'timestep', and 'flow' columns

# Get unique locations
locations = data['location'].unique()

for location in locations[:5]:
    location_data = data[data['location'] == location]

    # Extract X (independent variables) and y (dependent variable)
    X = location_data[['timestep', 'location']]
    y = location_data['flow']

    # 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 the KNN regression model
    n_neighbors = 5  # You can adjust the number of neighbors
```

Figure 6 shows an implementation of all ML models

5.2 DL Models

After this Deep learning models of CNN,LSTM and RNN were implemented with various test splits as mentioned above. Figure 7 shows implemtation of LSTM,RNN and CNN.

```
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error

# Assuming you have 'data' DataFrame loaded

# Select the first 5 locations
selected_locations = [0, 1, 2, 3, 4]

# Select relevant features (e.g., flow, time, speed, location)
selected_features = ['timestep', 'location', 'flow']

# Initialize dictionaries to store metrics for each location
r2_scores = {}
rmse_scores = {}
mae_scores = {}

for location in selected_locations:
    # Filter data for the current location
    location_data = data[data['location'] == location][selected_features]

    # Check if the filtered_data is empty
    if location_data.empty:
        raise ValueError(f"No data available for location {location}.")
```

```
# Combine normalized `X` and `y` for creating sequences
normalized_data = np.concatenate((X_scaled, y_scaled), axis=1)

# Create sequences for the LSTM model
def create_sequences(data, sequence_length):
    sequences = []
    for i in range(len(data) - sequence_length):
        sequence = data[i:i+sequence_length, :]
        target = data[i+sequence_length:i+sequence_length+1, -1]  # Assuming the target column is the last one ('
        sequences.append((sequence, target))
    return np.array([s[0] for s in sequences]), np.array([s[1] for s in sequences])

# Define sequence length
sequence_length = 10

# Create sequences
X, y = create_sequences(normalized_data, sequence_length)

# 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)

# Build the LSTM model with dropout and early stopping
model = Sequential()
model.add(LSTM(units=50, input_shape=(X_train.shape[1], X_train.shape[2])))  # Adjust input_shape if needed
model.add(Dense(units=1, activation='linear'))
model.compile(optimizer='adam', loss='mean_squared_error')
```

```
r2_scores = {}
rmse_scores = {}
mae_scores = {}

for location in selected_locations:
    # Filter data for the current location
    location_data = data[data['location'] == location][selected_features]

    # Check if the filtered_data is empty
    if location_data.empty:
        raise ValueError(f"No data available for location {location}.")

    # Separate scaler for the 'flow' feature
    flow_scaler = MinMaxScaler()
    y_scaled = flow_scaler.fit_transform(location_data['flow'].values.reshape(-1, 1))

    # Normalize the data for other features
    scaler = MinMaxScaler()
    X_scaled = scaler.fit_transform(location_data.drop('flow', axis=1))

    # Combine normalized 'X' and 'y' for creating sequences
    normalized_data = np.concatenate((X_scaled, y_scaled), axis=1)

    # Create sequences for the RNN model
    def create_sequences(data, sequence_length):
        sequences = []
        for i in range(len(data) - sequence_length):
```

Figure 7 shows implementation of CNN,LSTM and RNN.

After the implementation of DL models Statistical Models were implemented with the same train test split mentioned above.Figure 8 shows implementation of ARIMA and sarima Model

5.3 Statistical Models

```
# Extract and preprocess the data
y = location_data['flow']

# Split the data into training and testing sets (e.g., 80% for training, 20% for testing)
train_size = int(len(y) * 0.8)
train, test = y[:train_size], y[train_size:]

# Hyperparameter tuning using a grid search
best_r2 = -np.inf
best_params = None

for p in range(3):  # Adjust the range based on your dataset characteristics
    for d in range(2):  # Adjust the range based on your dataset characteristics
        for q in range(3):  # Adjust the range based on your dataset characteristics
            order = (p, d, q)
            model = ARIMA(train, order=order)
            model_fit = model.fit()
            y_pred = model_fit.predict(start=len(train), end=len(train) + len(test) - 1)
            r2 = r2_score(test[d:], y_pred[d:])
            if r2 > best_r2:
                best_r2 = r2
                best_params = order

# Train the ARIMA model with the best hyperparameters
best_model = ARIMA(y, order=best_params)
```

```
# Split the data into training and testing sets
train_size = int(len(y) * 0.8)
train, test = y[:train_size], y[train_size:]

# Hyperparameter tuning using itertools.product
p_values = range(3)
d_values = range(2)
q_values = range(3)

best_r2 = -np.inf
best_params = None

for order in product(p_values, d_values, q_values):
    try:
        model = SARIMAX(train, order=order, seasonal_order=(1, 0, 1, 12), enforce_stationarity=False, enforce_invertil
        model_fit = model.fit(disp=False, method='powell')
        y_pred = model_fit.predict(start=len(train), end=len(train) + len(test) - 1)
        r2 = r2_score(test, y_pred)
        if r2 > best_r2:
            best_r2 = r2
            best_params = order
    except Exception as e:
        print(f"Error fitting model with order {order}: {e}")

# Train the SARIMA model with the best hyperparameters
try:
```

Figure 8 shows implementation of the SARIMA and ARIMA models.

6. Results

6.1 Results of ML Models

After implementation, we would like to look on how our models performed. First we will look at how ML models performed.Fig 9 shows peromance of ML models.

```
Location 0:
R-squared (R2): 0.9867
Root Mean Squared Error (RMSE): 15.93
Mean Absolute Error (MAE): 5.3792
Location 1:
R-squared (R2): 0.9855
Root Mean Squared Error (RMSE): 18.46
Mean Absolute Error (MAE): 5.9633
Location 2:
R-squared (R2): 0.9999
Root Mean Squared Error (RMSE): 1.355
Mean Absolute Error (MAE): 0.1070
Location 3:
R-squared (R2): 0.9398
Root Mean Squared Error (RMSE): 38.28
Mean Absolute Error (MAE): 11.9697
Location 4:
R-squared (R2): 0.9957
Root Mean Squared Error (RMSE): 11.33
Mean Absolute Error (MAE): 3.1615
```

```
Location 0:
R-squared (R2): 0.9641
Root Mean Squared Error (RMSE): 26.1887
Mean Absolute Error (MAE): 19.3225
Location 1:
R-squared (R2): 0.9674
Root Mean Squared Error (RMSE): 27.4607
Mean Absolute Error (MAE): 20.5887
Location 2:
R-squared (R2): 0.9333
Root Mean Squared Error (RMSE): 30.6728
Mean Absolute Error (MAE): 21.8049
Location 3:
R-squared (R2): 0.9721
Root Mean Squared Error (RMSE): 25.8610
Mean Absolute Error (MAE): 19.4256
Location 4:
R-squared (R2): 0.9628
Root Mean Squared Error (RMSE): 33.3146
Mean Absolute Error (MAE): 24.4569
```

```
Location 0 - Gradient Boosting Regression (XGBoost):
R-squared (R2): 0.9325
Root Mean Squared Error (RMSE): 35.9019
Mean Absolute Error (MAE): 28.1385
Location 1 - Gradient Boosting Regression (XGBoost):
R-squared (R2): 0.9255
Root Mean Squared Error (RMSE): 41.4971
Mean Absolute Error (MAE): 33.1441
Location 2 - Gradient Boosting Regression (XGBoost):
R-squared (R2): 0.8928
Root Mean Squared Error (RMSE): 38.8785
Mean Absolute Error (MAE): 29.5506
Location 3 - Gradient Boosting Regression (XGBoost):
R-squared (R2): 0.9397
Root Mean Squared Error (RMSE): 38.0564
Mean Absolute Error (MAE): 30.5465
Location 4 - Gradient Boosting Regression (XGBoost):
R-squared (R2): 0.9361
Root Mean Squared Error (RMSE): 43.6674
Mean Absolute Error (MAE): 33.7548
```

```
location: 0.0000
Location 1:
R-squared (R2): 0.9954
Root Mean Squared Error (RMSE): 10.4112
Mean Absolute Error (MAE): 7.5612
Feature Importances:
timestep: 1.0000
location: 0.0000
Location 2:
R-squared (R2): 0.9899
Root Mean Squared Error (RMSE): 11.9876
Mean Absolute Error (MAE): 8.4947
Feature Importances:
timestep: 1.0000
location: 0.0000
Location 3:
R-squared (R2): 0.9958
Root Mean Squared Error (RMSE): 10.1055
Mean Absolute Error (MAE): 7.4088
Feature Importances:
timestep: 1.0000
location: 0.0000
Location 4:
R-squared (R2): 0.9947
Root Mean Squared Error (RMSE): 12.5811
Mean Absolute Error (MAE): 9.0733
```

Figure 9 shows results for decision tree,knn , gradient boost, and Random forest.

6.2 Results of DL Models

After this Deep learning results are displayed OF LSTM,CNN and RNN.Figure 10 shows the results of deep learning models of LSTM,CNN and RNN .

```
Epoch 56/100
154/154 [==============================] - 2s 11ms/step - loss: 0.0027 - val_loss: 0.0027
Epoch 57/100
154/154 [==============================] - 2s 10ms/step - loss: 0.0027 - val_loss: 0.0028
Epoch 58/100
154/154 [==============================] - 2s 11ms/step - loss: 0.0027 - val_loss: 0.0027
Epoch 59/100
154/154 [==============================] - 2s 10ms/step - loss: 0.0027 - val_loss: 0.0027
39/39 [==============================] - 1s 4ms/step

Evaluation metrics for Location 0:
R2 Score: 0.9580
RMSE: 28.3367
MAE: 21.4126
Epoch 1/100
154/154 [==============================] - 4s 13ms/step - loss: 0.0073 - val_loss: 0.0039
Epoch 2/100
154/154 [==============================] - 1s 10ms/step - loss: 0.0039 - val_loss: 0.0036
```

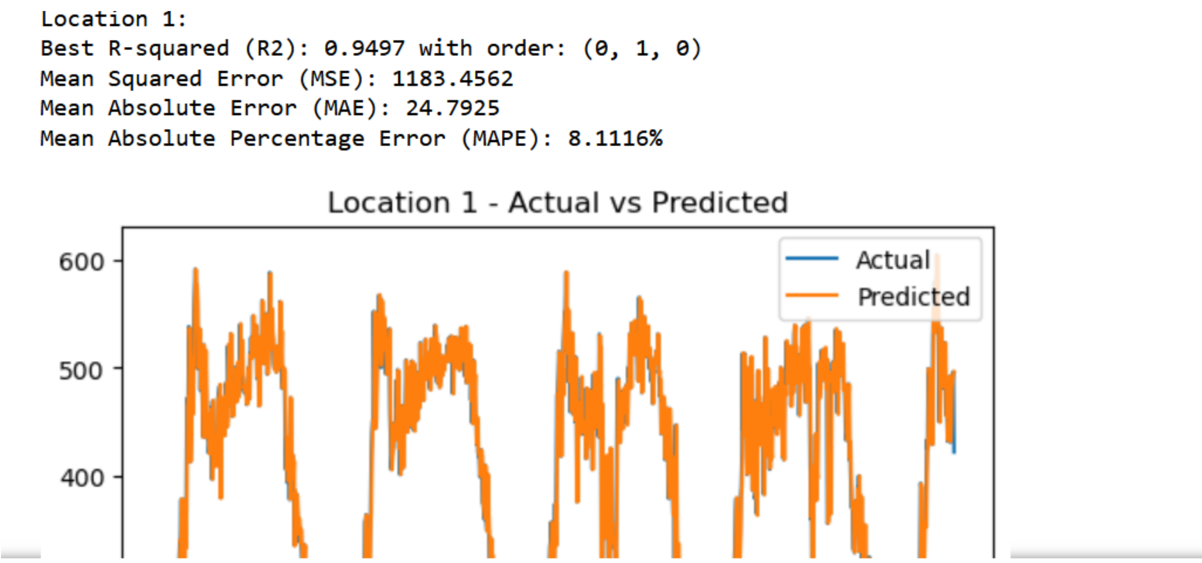
```
Overall Evaluation Metrics:
Location 0:
R2 Score: 0.9031955590222258
Root Mean Squared Error (RMSE): 43.040184012033734
Mean Absolute Error (MAE): 33.71461082220851
------------------------------
Location 1:
R2 Score: 0.9230105692195514
Root Mean Squared Error (RMSE): 42.43369615117888
Mean Absolute Error (MAE): 32.770176280628554
------------------------------
Location 2:
R2 Score: 0.8750020148410159
Root Mean Squared Error (RMSE): 42.167681411887195
Mean Absolute Error (MAE): 30.913365023476736
------------------------------
Location 3:
R2 Score: 0.9172016450914084
```

```
Location 1:
R2 Score: 0.9623830049973586
Root Mean Squared Error (RMSE): 29.66110714082697
Mean Absolute Error (MAE): 22.566484163333843
------------------------------
Location 2:
R2 Score: 0.9124354734213089
Root Mean Squared Error (RMSE): 35.29330405750933
Mean Absolute Error (MAE): 25.43836583719625
------------------------------
Location 3:
R2 Score: 0.9661503215591642
Root Mean Squared Error (RMSE): 28.572839976796043
Mean Absolute Error (MAE): 21.69661939298952
------------------------------
Location 4:
R2 Score: 0.9571377299784966
Root Mean Squared Error (RMSE): 35.68937955041154
```

Figure 10 shows the results of deep learning models of LSTM, CNN and RNN .

6.3 Results of Statistical Models

After this checked results of the ARIMA and SARIMA mode.Figure 11 shows graph and a result of   SARIMA and AARIMA



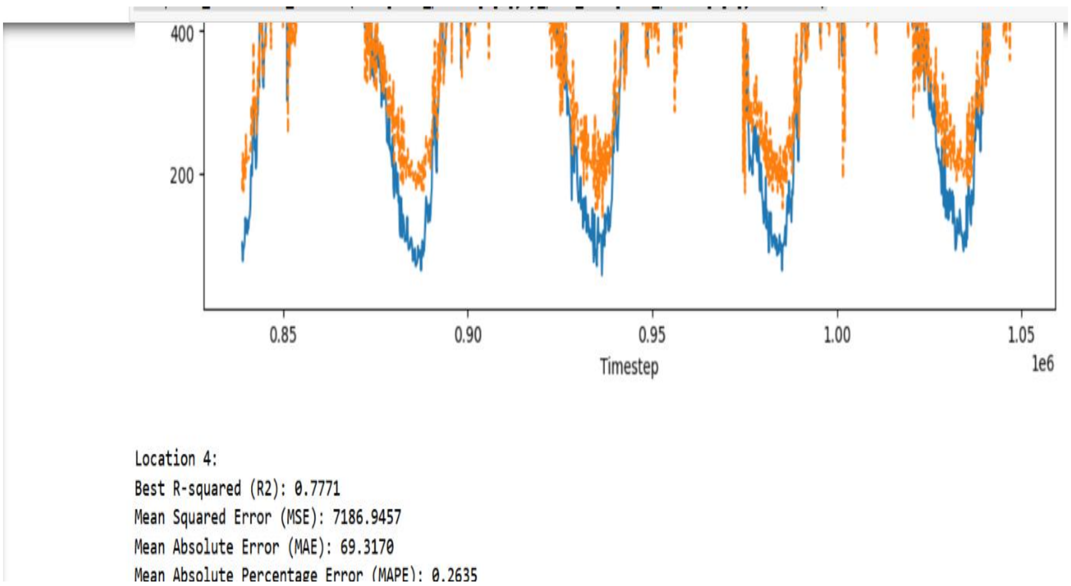


Figure 11 shows results of ARIMA and SARIMA models.

6.4 K cross Validation

After this Kcross validation on ML and DL models was performed, Figure 12 shows implementation of kcross validation of ML and DL models.

```python
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import LinearRegression
from xgboost import XGBRegressor  # Import XGBRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Set the number of folds for cross-validation
k = 5  # Adjust as needed

# Assuming you have a DataFrame called 'data' with 'location', 'timestep', and 'flow' columns

# Get unique locations
locations = data['location'].unique()

# Define the models
models = {
    'Random Forest': RandomForestRegressor(n_estimators=100),
    'K-Nearest Neighbors': KNeighborsRegressor(n_neighbors=5),
    'Decision Tree': DecisionTreeRegressor(),
    'XGBoost': XGBRegressor(objective='reg:squarederror')  # Specify objective for regression
}

# Define the features and target variable
features = ['timestep', 'location']
target = 'flow'
```

```python
# Initialize dictionaries to store metrics for each location and model
r2_scores_lstm = {}
rmse_scores_lstm = {}
mae_scores_lstm = {}

r2_scores_rnn = {}
rmse_scores_rnn = {}
mae_scores_rnn = {}

r2_scores_cnn = {}
rmse_scores_cnn = {}
mae_scores_cnn = {}

# Set up k-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

for location in selected_locations:
    # Filter data for the current location
    location_data = data[data['location'] == location][selected_features]

    # Check if the filtered_data is empty
    if location_data.empty:
        raise ValueError(f"No data available for location {location}.")

    # Separate scaler for the 'flow' feature
```

Figure 12 shows implementation of kcross validation of ML and DL models.

6.5 K cross validation Results

Results were displayed of Kcross validation of ML and DL models and were very positive.Figure 13 shows k cross-validation results of ML and DL models..

```
Average R-squared (R2):0.9252
Average Root Mean Squared Error (RMSE): 32.6393
Average Mean Absolute Error (MAE): 23.4638
Location 3:
Average R-squared (R2):0.9686
Average Root Mean Squared Error (RMSE): 27.6034
Average Mean Absolute Error (MAE): 20.5638
Location 4:
Average R-squared (R2):0.9592
Average Root Mean Squared Error (RMSE): 34.8986
Average Mean Absolute Error (MAE): 25.1836

Cross-validation results for K-Nearest Neighbors:

Location 0:
Average R-squared (R2):0.9661
Average Root Mean Squared Error (RMSE): 25.4110
Average Mean Absolute Error (MAE): 19.1112
Location 1:
Average R-squared (R2):0.9674
Average Root Mean Squared Error (RMSE): 27.6836
Average Mean Absolute Error (MAE): 20.3702
Location 2:
Average R-squared (R2):0.9293
Average Root Mean Squared Error (RMSE): 31.7273
```

```
Location 0 LSTM Metrics:
Average R2 Score: 0.9583450115763135
Average Root Mean Squared Error (RMSE): 28.15854085087896
Average Mean Absolute Error (MAE): 21.13679860843072
-------------------------
Location 0 SimpleRNN Metrics:
Average R2 Score: 0.9071402023821028
Average Root Mean Squared Error (RMSE): 42.05714937770291
Average Mean Absolute Error (MAE): 32.81703400608872
-------------------------
Location 0 CNN Metrics:
Average R2 Score: 0.9573618476625608
Average Root Mean Squared Error (RMSE): 28.484445598909826
```

Figure 13 shows k cross-validation results of ML and DL models..