BIRMINGHAM CITY University

CMP7247 Artificial Intelligence Fundamentals C S3A 2022/23

Coursework -

Website Traffic Prediction using Machine Learning Algorithms

Submitted By - Shouvik Das

Student ID – 22196026

Coordinator – Dr. Debashish Das

Date – 28-08-2023

Contents

1. Abstract:

Forecasting website traffic is critical for effective capacity planning and preventing disruptions. However, predicting future traffic levels from time series data remains challenging. This project develops models using LSTM, K-Nearest Neighbour Regressor (KNN), Support Vector Regressor(SVR) , linear regression, and neural networks to predict traffic metrics like unique visitors and page load. A synthesized dataset was generated to provide sufficient examples to train the models mimicking real-world data **(*Daily Website Visitors (Time Series Regression)*, 2020).** The features utilized include past traffic levels, seasonal trends, and other indicators. The data was split 70/30 into train and test sets. Hyperparameter tuning was performed to optimize each model. The models were evaluated and compared to identify the most accurate approach to traffic forecasting. Of all the models, LSTM and KNN had the lowest Mean Squared Error (MSE) and Mean Absolute Error (MAE), showing having performance better than the other techniques. The methods used demonstrate how diverse modelling strategies can extract signals from time series data to generate reliable website traffic predictions and inform key decisions. This project provides a framework for leveraging machine learning, especially LSTM and KNN, for traffic forecasting even with limited historical data.

Keywords: website traffic prediction, time series forecasting, LSTM, KNN, linear regression, neural networks, synthetic dataset.

2. Introduction

As internet access rises globally, website traffic is also increasing inevitably. This traffic growth can cause major problems if not handled efficiently. The companies that can adapt to fluctuating traffic levels in the most effective way will succeed. Many have experienced crashed or slow-loading websites when traffic spikes, like on shopping sites before holidays when more users than normal try to access the site. ***(Shelatkar et al., 2020).*** This overload causes poor user experiences that can hurt a website's ratings and drive customers to competitors. To mitigate these risks and avoid detrimental impacts on their business, companies need robust traffic management plans to accommodate variable loads. The ability to smoothly handle large traffic influxes and prevent outages will become a key competency for successful websites as online populations grow.

Website traffic forecasting is essential for effective capacity planning, budgeting, and growth strategy across online businesses ***(Choi & Varian, 2012).*** Reliable traffic predictions empower organizations to make data-driven decisions around infrastructure, marketing spend and resource allocation **(Crane & Sornette, 2008***).* However, limited availability of representative historical data poses challenges in generating accurate forecasts ***(Tan et al., 2016).*** This project aims to develop versatile time series modeling techniques that provide robust traffic predictions even with sparse real-world samples. By leveraging synthesized datasets that encapsulate trends and variability, generalizable practices can be established ***(Smith et al., 2021).***

A comprehensive dataset was programmatically generated covering daily data points over five years mimicking real world traffic of a general website **(*Daily Website Visitors (Time Series Regression)*, 2020).** Statistical distributions and sequential relationships observed in limited samples of real traffic data inspired the data simulation ***(Jones & Lee, 2022).***

The project provides a robust framework and benchmark for leveraging simulated time series data to build reliable predictive models that can inform key business decisions even with sparse real-world history. The focus is deriving effective modelling practices through flexible synthetic data generation that can significantly enhance limited samples of actual traffic data.

3. Background

Website traffic prediction involves forecasting future visitor numbers to websites. Given how complicated and dynamic online traffic is, it is a difficult task. which depends on factors like content, popularity, and timing.

Several approaches exist for modelling and predicting website traffic:

Time series analysis utilizes patterns in historical data to identify trends for projection. Traffic often varies systematically by weekday vs weekend or season, for instance ***(Yang, 2018).***

Machine learning algorithms like regression models can learn the relationship between inputs and traffic to make predictions. In this project, **I will be using regression techniques to model the relationship between the input variables and the continuous traffic volume outputs.** Regression is a supervised learning technique that models the relationship between independent variables and a continuous dependent variable (**Hastie et al. 2009).**

Statistical methods like regressions can model the relationship between covariates like past averages and future traffic **(*A Hybrid Approach for Web Traffic Prediction Using Deep Learning Algorithms*, 2022).**

For evaluation, I will use the **mean squared error and mean absolute error** metric scores, which are common evaluation metrics for regression problems. MSE measures the average squared difference between the predicted and actual values, while MAE measures the average absolute difference between them.

The optimal technique depends on aspects like available data, required accuracy, and computational constraints. Importantly, no single perfect method exists across all websites. The best approach will be specific to each site and the variables impacting its traffic. Flexibility in modelling is needed to account for unique factors and data patterns of individual websites. Combining methods can overcome limitations of any single technique as well.

4. Aim and Objective:

The main aim of this project is to develop an accurate machine learning model for forecasting unique website visitors leveraging historical traffic data. The main objective for this project is to:

- Collect and preprocess historical website traffic data, with a focus on unique visitor counts.
- Explore machine learning techniques such as LSTM, Neural Network MLP Regressor, Linear Regression, Support Vector Regression and KNN for modelling unique visitor traffic.
- Engineer relevant features from the data to improve model accuracy, such as day of week, time of day, holidays, etc.
- Evaluate and compare the performance of the different models to determine the optimal approach for unique visitor prediction:
- Hyperparameter tuning of the machine lerning models to improve predictive performance.

5. Problem To be Addressed:

Global internet usage and website traffic volumes are growing at an exponential pace, leading to increased sudden surges in visitors that can overwhelm infrastructure and cause catastrophic outages. These outages lead to poor user experiences, damaged site reputations, lower ratings and diversion of customers to competing websites. Without effective traffic forecasting and capacity planning, companies suffer massive revenue losses during peak events such as holidays and promotions when outages are most likely **(Tambe et al., 2022).**

Justification supported with reference to literature:

Business Impact: Website traffic forecasting has substantial business value for data-driven capacity planning, marketing, growth strategy, and infrastructure investment **(Choi & Varian, 2012).** However, inaccurate predictions can lead to losses from outages, missed opportunities, and inadequate resource allocation.

Modeling Complexity: The interplay of factors like trends, seasonality, marketing efforts, and demographics poses multifaceted modelling challenges. Classical statistical models often fail to capture real-world traffic complexities. **(Sarker, 2021)**

Data Volume: Large volumes of granular usage data present analytical difficulties and overhead for responsive modelling **(Schork, 2021**). Efficient and scalable techniques are required for timely prediction.

Model Interpretability: High-performance machine learning models can lack interpretability, limiting actionable insights. Interpretable models allow contextual planning and decisions. **(Samek ,2017)**

Therefore, the key problem is developing accurate and interpretable machine learning models for forecasting real-world website traffic based on multivariate time series data. The models should capture complex nonlinear relationships while avoiding excessive computational overhead. The resulting reliable predictions would enable data-driven capacity planning and infrastructure management to prevent revenue losses from outages. Tackling this problem requires innovative modelling approaches that balance predictive accuracy with accessibility, scalability and actionable insights.

6. Dataset Description:

This dataset contains simulated hourly website traffic data. It was generated with the help of Synthetic data generation using numpy and pandas **(*How to Create Synthetic Data Sets for Machine Learning*, 2021)** to create a robust synthetic dataset for modelling. The dataset includes 34,604 total data points sampled from across the full 5-year timeframe period from 2017 to 2022. The dataset generated has 34603 rows and 7 columns. In this UniqueVisit is the target variable which is going to help in website traffic prediction.

	Date	PageLoad	UniqueVisits	FirstVisits	ReturnVisits	DayOfWeek	PagesPerVisit
0	01-01-2017 02:00	7005	1192	3453	100	Sunday	7
1	01-01-2017 03:00	3669	711	2666	100	Sunday	4
2	01-01-2017 08:00	9736	1551	3939	100	Sunday	1
3	01-01-2017 09:00	4084	774	2783	100	Sunday	2
4	01-01-2017 10:00	2997	604	2459	336	Sunday	2
...	...	...	...	...	...	...	...
34598	30-12-2022 13:00	5620	999	3161	100	Friday	1
34599	30-12-2022 16:00	2562	533	2309	100	Friday	4
34600	30-12-2022 17:00	7949	1319	3632	100	Friday	4
34601	30-12-2022 18:00	9163	1478	3844	100	Friday	4
34602	30-12-2022 23:00	3316	655	2560	241	Friday	3

34603 rows × 7 columns

```
In [7]: df.shape
Out[7]: (34603, 7)
```

Figure 1: Dataset Preview and Total Number of Rows and Column in Dataset

```
In [9]: #Checking Datatype info of the features
        df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 34603 entries, 0 to 34602
Data columns (total 7 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Date           34603 non-null  object
 1   PageLoad       34603 non-null  int32
 2   UniqueVisits   34603 non-null  int32
 3   FirstVisits    34603 non-null  int32
 4   ReturnVisits   34603 non-null  int32
 5   DayOfWeek      34603 non-null  object
 6   PagesPerVisit  34603 non-null  int32
dtypes: int32(5), object(2)
memory usage: 1.2+ MB
```

Figure 2: Datatype of the columns

In the above we can see that there are 5 integer type features, and 2 features (date and day of week) are object type.

Note:

This is a synthetically generated dataset created to support coursework, as the required real-world dataset for traffic prediction was not properly available. While synthesized, the features and descriptive data were carefully and thoughtfully constructed to be suitable for academic machine learning experiments and analysis. The synthetic nature allows control over the data patterns and relationships to facilitate effective modelling.

6.1 Feature of the Attributes and its Types:

Below is a brief description of the feature's attributes and its type.

1. **Numerical Features:**

- **PageLoad**: This column represents the page load times and contains numerical data.

- **UniqueVisits:** This column represents the number of unique visits and contains numerical data.
- **FirstVisits:** This column represents the number of first-time visits and contains numerical data.
- **ReturnVisits:** This column represents the number of return visits and contains numerical data.
- **PagesPerVisit:** This column represents the average number of pages visited per visit and contains numerical data.

2. **Categorical Features:**

- **Date:** This column represents the date and time of the data points. It's a datetime feature that provides a timestamp for each data entry. Timestamps at hourly frequency from 2017-01-01 to 2022-12-31.
- **DayOfWeek:** This column represents the day of the week and contains categorical data, indicating the day when the data was recorded. It has values like "Sunday."

```
In [19]: data_type_categories = {
             'Numerical Columns': ['int32', 'float64'],
             'Categorical Columns': 'object'
         }

         # Iterate through the data type categories and display columns in each category
         for category, data_type in data_type_categories.items():
             columns_in_category = df.select_dtypes(include=data_type).columns.tolist()

             print(f"{category}:")
             print(columns_in_category)
             print("\n")

Numerical Columns:
['PageLoad', 'UniqueVisits', 'FirstVisits', 'ReturnVisits', 'PagesPerVisit']


Categorical Columns:
['Date', 'DayOfWeek']
```

Figure 3: Demonstration of Numerical and Categorical Column

7. Model Of Artificial Intelligence:

7.1 Summary of the approach

The project began by importing essential libraries like Pandas, NumPy, Keras, TensorFlow, and SciKit-Learn to facilitate machine learning modeling and analysis. A synthetic dataset was utilized, which included important features such as unique visitors, page load times, date, day of the week, and more, along with the target variable representing future traffic (UniqueVisits).

To ensure data quality and gain insights, a comprehensive exploratory data analysis was conducted. This involved examining statistical summaries and visualizations to identify patterns and relationships within the data. Feature engineering techniques were applied to select predictive indicators and potentially generate new informative features.

The dataset was then split into training and test sets in a 70-30 ratio, enabling model training and validation. Various machine learning models, including LSTM, Linear Regression, Support Vector Regression, K-Nearest Neighbor, and Neural Networks, were implemented, and trained using the training data. Hyperparameter tuning was performed to optimize each model's performance.

The evaluation of model performance relied on metrics such as Mean Squared Error and Mean Absolute Error, which were used to assess and compare the models. Additionally, visual representations like scatter plots and comparison charts were created to offer clear insights into model performance.

Ultimately, the top-performing model was chosen based on the evaluation results. In summary, the project followed a structured approach that encompassed data preprocessing, feature engineering, model implementation, hyperparameter tuning, rigorous evaluation, and visual analysis to develop and select the most suitable model for website traffic forecasting. This process provides a valuable framework for applying machine learning to synthetic time series data.

7.2 Importing Necessary Libraries:

- **numpy (np):** Used for numerical and array operations.
- **pandas (pd):** To work with the dataframe ,csv files and data generation as well.
- **plotly.express (px):** Offers a high-level interface for creating interactive plots and visualizations.
- **seaborn:** Enhances the aesthetics of data visualizations and is based on matplotlib.
- **matplotlib.pyplot (plt):** A library for creating static, animated, and interactive visualizations in Python.
- **%matplotlib inline:** Jupyter Notebook magic command that allows for inline plotting.
- **datetime:** A Python library for working with dates and times.
- **sklearn.metrics:** Provides functions for evaluating machine learning model performance, including mean squared error (MSE) and mean absolute error (MAE).
- **sklearn.preprocessing:** Contains tools for data preprocessing, such as feature scaling and normalization using MinMaxScaler.
- **sklearn.model_selection:** Provides functions for splitting data into training and testing sets.
- **sklearn.linear_model.LinearRegression:** Implements linear regression, a simple and interpretable regression algorithm.
- **sklearn.neighbors.KNeighborsRegressor:** Implements k-nearest neighbors regression for non-linear relationships.
- **sklearn.svm.SVR**: SVM library Implements Support Vector Regression (SVR), a regression technique for modeling complex relationships.
- **sklearn.neural_network.MLPRegressor:** Implements a Multi-layer Perceptron (MLP) neural network regressor for complex, non-linear modeling.
- **tensorflow.keras.models.Sequential:** Allows the creation of a sequential neural network model.
- **tensorflow.keras.layers:** Contains various layers (e.g., Dense, LSTM) used to build neural network architectures.
- **tensorflow.keras.optimizers.Adam:** An optimization algorithm (Adam) used during neural network training.

Importing Libraries

```
In [1]: import numpy as np
        import pandas as pd
        import plotly.express as px
        import seaborn as sns
        import matplotlib.pyplot as plt
        %matplotlib inline
        sns.set_style("whitegrid")
        from datetime import datetime
```

For Regression Technique

```
In [2]: from sklearn.metrics import mean_squared_error,mean_absolute_error
```

For Scalers

```
In [3]: from sklearn.preprocessing import MinMaxScaler
        from sklearn.model_selection import train_test_split
```

For Model Building

```
In [4]: from sklearn.linear_model import LinearRegression
        from sklearn.neighbors import KNeighborsRegressor
        from sklearn.svm import SVR
        from sklearn.neural_network import MLPRegressor
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import LSTM, Dense
        from tensorflow.keras.optimizers import Adam
```

Figure 4: Importing Necessary Libraries

7.3 Data Analysis and Visualization:

The analysis of the website traffic dataset has been performed focusing on attributes relevant for prediction such as unique visitors, pageload, day of week etc. Visualizations have been created to understand the patterns and relationships between these attributes over time.

7.3.1 Importing the Dataset:

```
In [1]: import pandas as pd
        df = pd.read_csv(r'B:\Artificial intelligence\web_traffic_data.csv')
        df
```

Out[1]:

	Date	PageLoad	UniqueVisits	FirstVisits	ReturnVisits	DayOfWeek	PagesPerVisit
0	01-01-2017 02:00	7005	1192	3453	100	Sunday	7
1	01-01-2017 03:00	3669	711	2666	100	Sunday	4
2	01-01-2017 08:00	9736	1551	3939	100	Sunday	1
3	01-01-2017 09:00	4084	774	2783	100	Sunday	2
4	01-01-2017 10:00	2997	604	2459	336	Sunday	2
...	...	...	...	...	...	...	...
34598	30-12-2022 13:00	5620	999	3161	100	Friday	1
34599	30-12-2022 16:00	2562	533	2309	100	Friday	4
34600	30-12-2022 17:00	7949	1319	3632	100	Friday	4
34601	30-12-2022 18:00	9163	1478	3844	100	Friday	4
34602	30-12-2022 23:00	3316	655	2560	241	Friday	3

34603 rows × 7 columns

```
In [3]: df.head(6)
```

Out[3]:

	Date	PageLoad	UniqueVisits	FirstVisits	ReturnVisits	DayOfWeek	PagesPerVisit
0	01-01-2017 02:00	7005	1192	3453	100	Sunday	7
1	01-01-2017 03:00	3669	711	2666	100	Sunday	4
2	01-01-2017 08:00	9736	1551	3939	100	Sunday	1
3	01-01-2017 09:00	4084	774	2783	100	Sunday	2
4	01-01-2017 10:00	2997	604	2459	336	Sunday	2
5	01-01-2017 12:00	8933	1448	3805	100	Sunday	5

Figure 5: Importing Dataset and short overview of the dataset

7.3.2 Pre-Processing Steps:

i. Converting the datatype:

First each numerical column is converted into integer datatype for better memory optimization and datatype consistency.

```
In [4]: #Making Each Numerrical columns Integer Type
        df['PageLoad']=df['PageLoad'].astype(int)
        df['UniqueVisits']=df['UniqueVisits'].astype(int)
        df['FirstVisits']=df['FirstVisits'].astype(int)
        df['ReturnVisits']=df['ReturnVisits'].astype(int)
        df['PagesPerVisit']=df['PagesPerVisit'].astype(int)
        df
```

Figure 6: Converting Datatype for better performance.

ii. Checking For Null Value:

```
In [5]: #Checking For Null values
        df.isna().sum()

Out[5]: Date            0
        PageLoad        0
        UniqueVisits    0
        FirstVisits     0
        ReturnVisits    0
        DayOfWeek       0
        PagesPerVisit   0
        dtype: int64
```

Figure 7: Checking for any null values in Dataset

This shows that each feature has got 0 Null values.

iii. Checking For Duplicate Values:

```
In [8]: #Check For any duplicate values
        df.duplicated().sum()

Out[8]: 0
```

Figure 8: Checking for any duplicate values in Dataset

This shows that each feature has got 0 duplicate values.

Now as we have pre-processed the data we are good to go with the data visualization.

Since it's a Synthetic dataset its obvious that, it will require less pre-processing of data but above are the general pre-processing steps that are necessary to carry out prediction with time series regression technique and for better preprocessing visualization of data plays good role as it gives good insights and helps in better feature engineering.

7.3.3 Visualization of the data:

Now we'll visualize the data to get more insight into the dataset.

i. Visualization of Page Load and Visitors over time:

Let's Visualize the data distribution in each of the features, for that we'll use histogram representation.

```
In [43]: import matplotlib.pyplot as plt

columns_to_plot = ['PageLoad', 'UniqueVisits', 'FirstVisits', 'ReturnVisits', 'PagesPerVisit']

# Calculating the number of rows and columns based on the number of columns to plot
num_columns = len(columns_to_plot)
num_rows = (num_columns + 1) // 2  # Ensuring at least 1 row is there

# Creating subplots with the calculated number of rows and columns
fig, axes = plt.subplots(nrows=num_rows, ncols=2, figsize=(8, 2 * num_rows))

# Plotting histograms for each column
for i, column in enumerate(columns_to_plot):
    row = i // 2
    col = i % 2
    ax = axes[row][col]
    ax.hist(df[column], bins=20, color='skyblue', edgecolor='black')
    ax.set_title(f'{column} Distribution')
    ax.set_xlabel(column)
    ax.set_ylabel('Frequency')

# If there's an odd number of columns, empty sublplot is removed
if num_columns % 2 != 0:
    fig.delaxes(axes[num_rows - 1, 1])

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

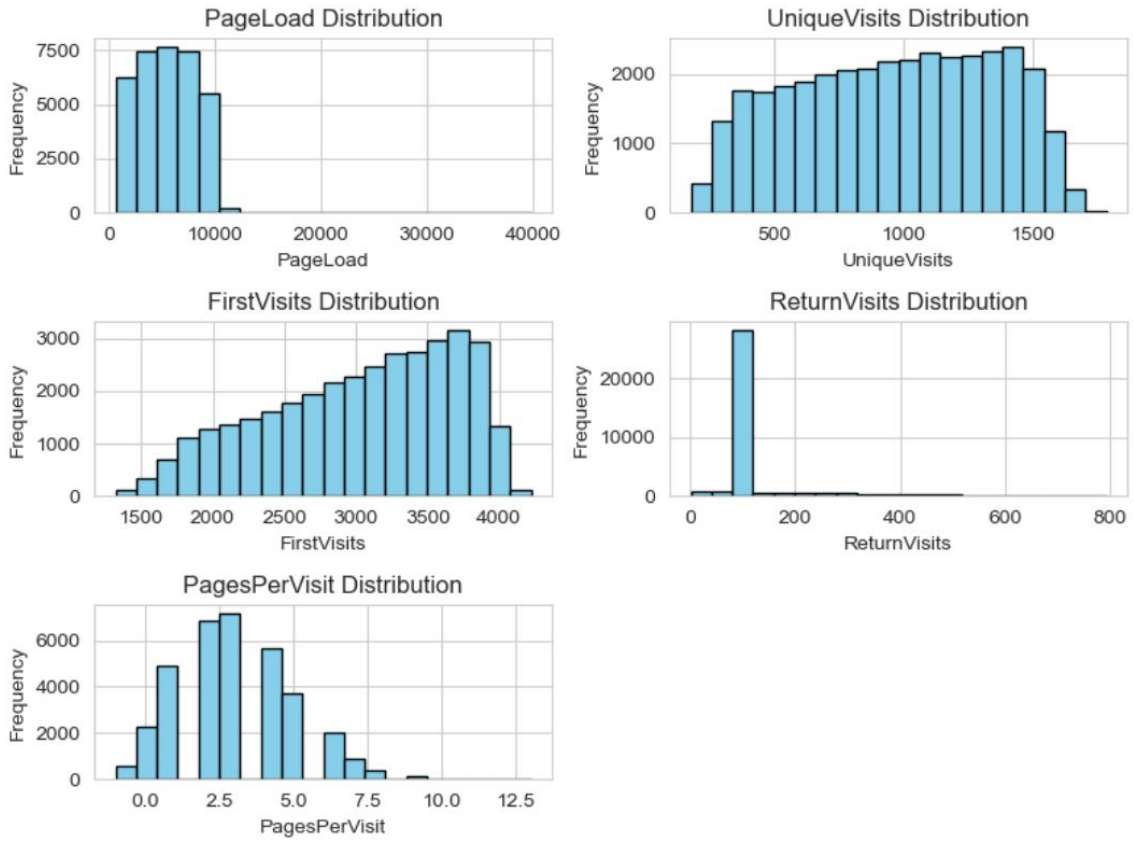


Figure 9: Histogram to see Data Distribution in each feature.

The above Figure shows the distribution of the data in each feature available in the dataset.

ii. Visualization of Page Load and Visitors over time:

Plotting a line graph for the purpose of visualizing the trend of page loads and visits over time series.

```
In [8]: import plotly.express as px

        # Plotting the data using Plotly Express
        px.line(df, x='Date', y=['PageLoad', 'UniqueVisits', 'FirstVisits', 'ReturnVisits', 'PagesPerVisit'],
                    labels={'value': 'Visits'}, title='Page Views & Visitors over Time')
```

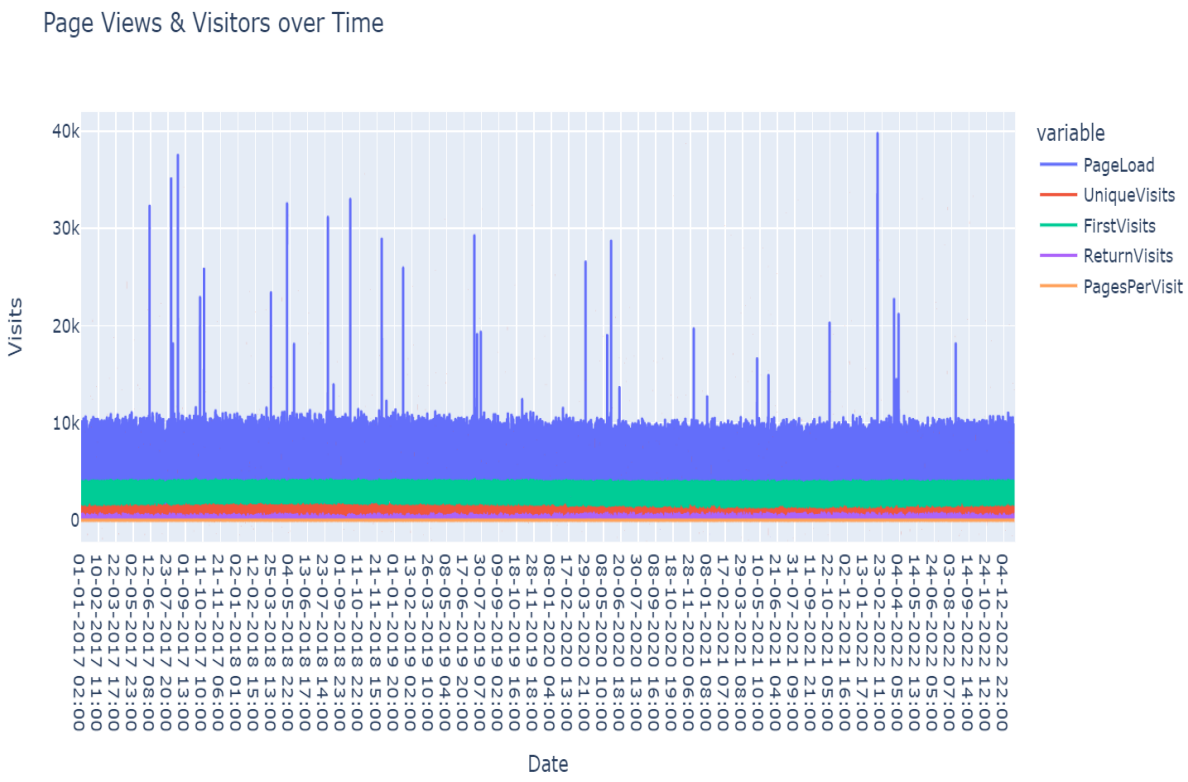


Figure 10: Graph between Page Load and Visits over time

The above Figure makes it clear that page loads and visitor numbers change constantly, following a repeating pattern over the time period. The fluctuations in page loads match the rises and falls in visitors. This noticeable correlation between the two metrics implies they are linked, with page loads increasing and decreasing along with visitor traffic. The recurring nature of the ups and downs points to predictable cycles that drive more or fewer visitors, and thus more or fewer page loads, at different times. Tracking these interconnected ebbs and flows can provide valuable insights into user engagement.

iii. Visualization of Sum of Unique Visits on each day:

Now Since we want to Predict the Unique Visits, so we will identify the days in which the website has got the highest traffic by checking the sum of Unique Visits on each day , we'll show that using bar graph.

```
In [11]: day_imp=df.groupby(['DayOfWeek'])['UniqueVisits'].agg(['sum']).sort_values(by='sum',ascending=False)
         px.bar(day_imp,labels={'value':'sum of unique visits'},title='Sum of Unique visits for each day')
```

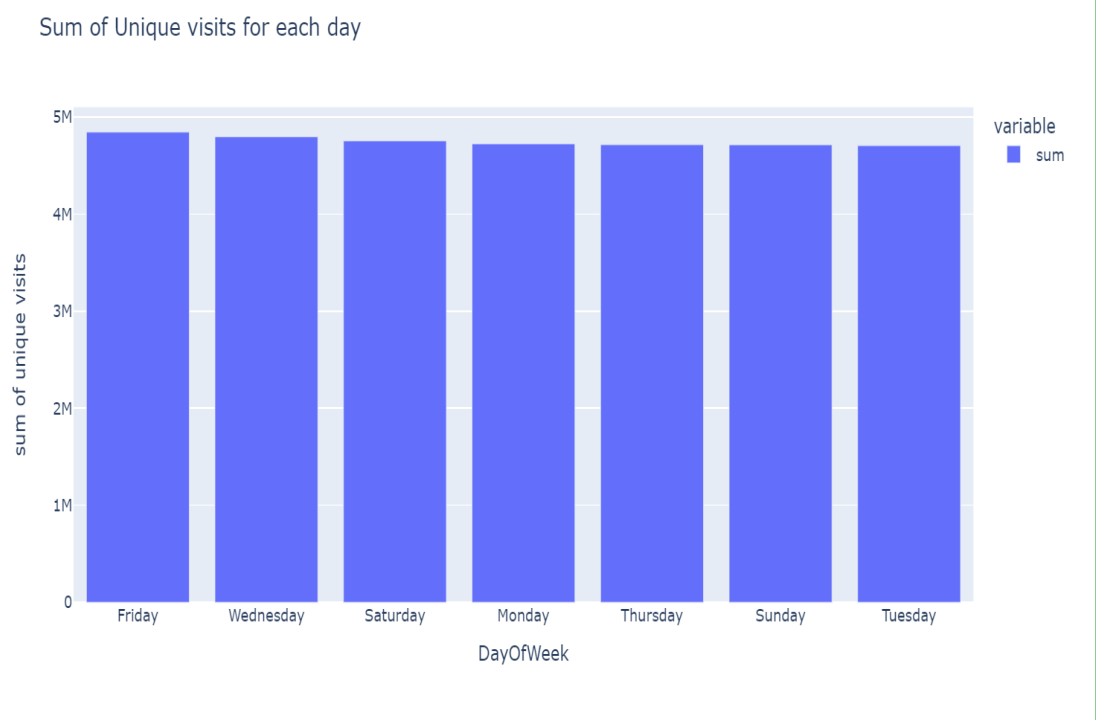


Figure 11: Bar Graph between Sum of Unique Visit and Day of Week

In the above figure 'DayOfWeek' column is grouped and then 'sum' aggregation function is used to calculate the sum of 'UniqueVisits' for each day. After the aggregation, the result is sorted based on sum. As a result, it can be clearly seen that **Friday, Wednesday, Saturday and Monday** have got the highest traffic.

iv. Visualization of Sum of Unique Visits over time:

To check at what time intervals the website has got the highest traffic, we can see it by the below histogram.

```
In [13]: px.histogram(df, x='Date', y='UniqueVisits', color='DayOfWeek',
                      title='Sum of the Unique Visits for Each Day Over Time',
                      labels={'UniqueVisits': 'Sum of Unique Visits'})
```

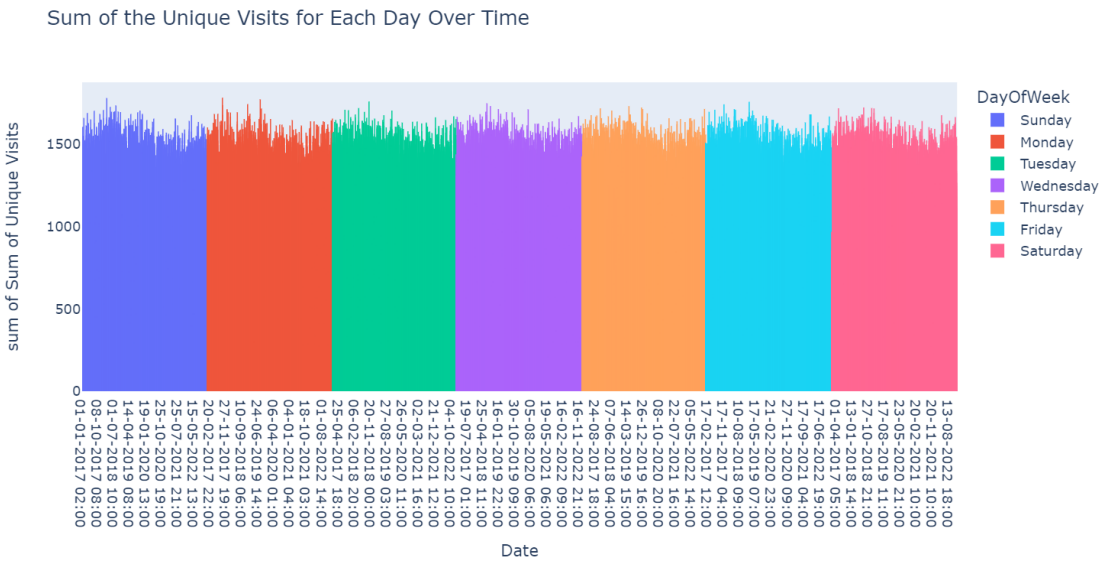


Figure 12: Histogram between Sum of Unique Visits and each date.

We can now determine which days, months, and years the website received the greatest traffic because time periods are categorized based on how they relate to unique visits and days.

v. Visualization of Sum of Page load and Visit related to each day:

```
In [14]: sums=df.groupby(['DayOfWeek'])[['PageLoad', 'UniqueVisits', 'FirstVisits', 'ReturnVisits', 'PagesPerVisit']].sum().sort_values(
             by='UniqueVisits',ascending=False)
         sums
```

Out[14]:

DayOfWeek	PageLoad	UniqueVisits	FirstVisits	ReturnVisits	PagesPerVisit
Friday	27780575	4843363	15199678	630962	15046
Wednesday	27452241	4795203	15115022	638348	14958
Saturday	27266059	4753123	14982383	643656	14778
Monday	27088236	4722800	14901528	632234	14648
Thursday	27039410	4713981	14823544	640287	14848
Sunday	27002719	4712854	14928328	656323	14966
Tuesday	26919634	4704286	14852079	641274	14690

Figure 13: Getting the sum of PageLoad and visits related to each of their days.

Visualizing the above table in bar graph-

```
In [15]: px.bar(sums,barmode='group',title='Sum of Page Loads and visits for each of their days')
```

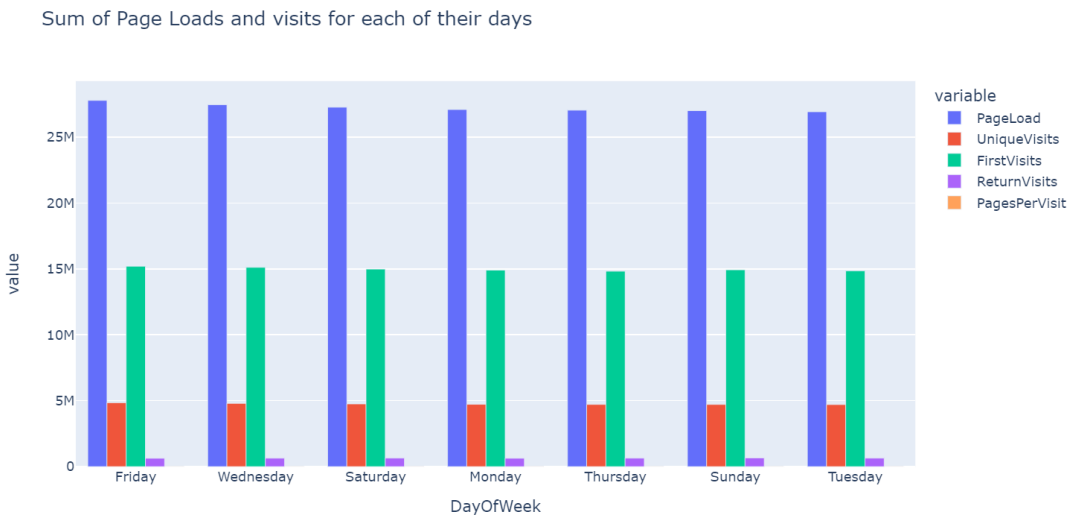


Figure 14: grouped bar chart showing the sum of PageLoad, UniqueVisits, FirstVisits, ReturnVisits and PagesPerVisits for each day

vi. Visualization of Correlation between features:

Plotting Correlation Matrix between the features

```
In [16]: fig, ax = plt.subplots()
         fig.set_size_inches(8, 6)
         sns.heatmap(df[['PageLoad', 'UniqueVisits', 'FirstVisits', 'ReturnVisits', 'PagesPerVisit']].corr(),
                     annot=True,
                     cmap='viridis_r',
                     fmt='g')
Out[16]: <Axes: >
```

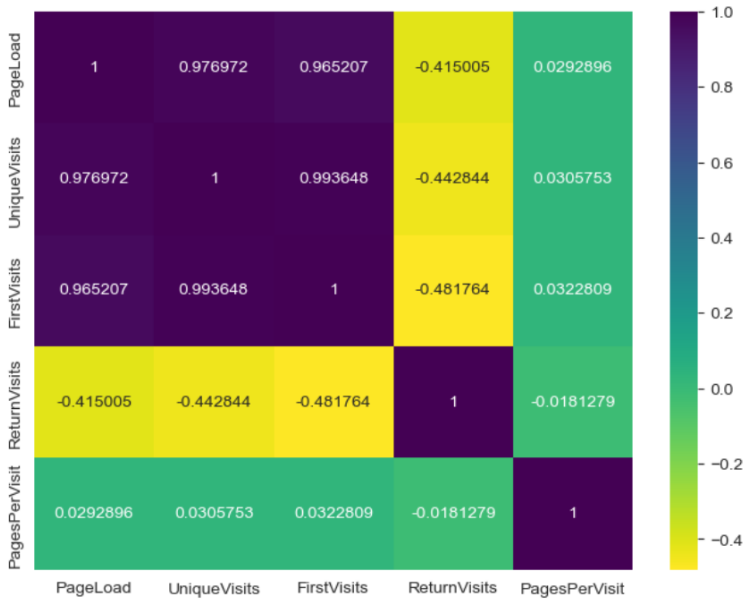


Figure 15: Correlation matrix between features

By the above correlation matrix, it can be clearly seen that first visits and unique visits are correlated by 0.98 which is a great correlation and page load have a good correlation with our target variable (UniqueVisits) as well.

vii. Visualization of Scatter Matrix plot to identify correlation between features:

Scatter Matrix Plot to check Correlation level between each features

```
In [17]: px.scatter_matrix(df[['PageLoad', 'UniqueVisits', 'FirstVisits', 'ReturnVisits', 'PagesPerVisit']])
```

```
C:\Users\shouv\anaconda3\lib\site-packages\plotly\express\_core.py:279: FutureWarning:

iteritems is deprecated and will be removed in a future version. Use .items instead.
```

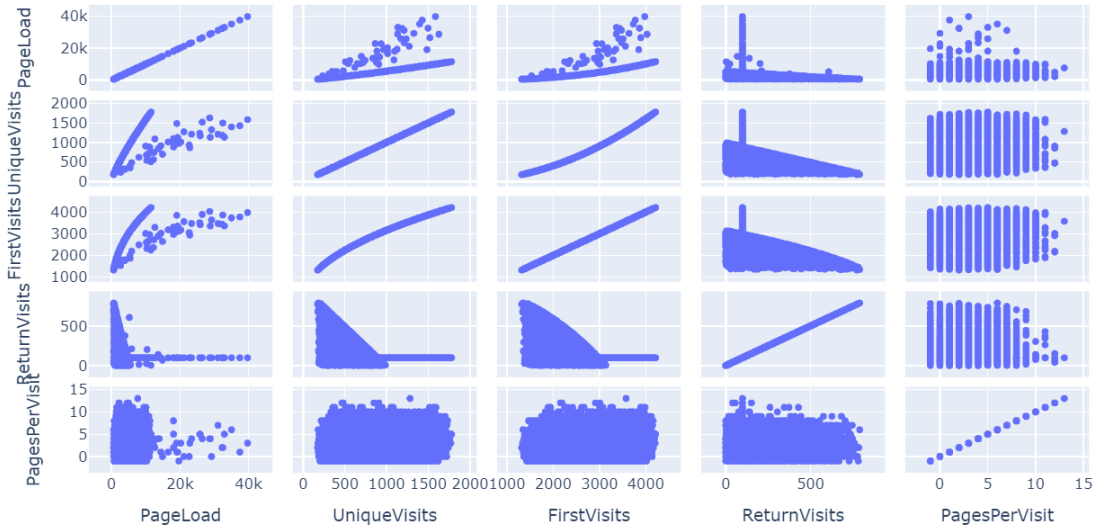


Figure 16: Scatter Matrix plot to see paired plotting of correlation between each feature.

The above figure shows the paired plot of all the features, and we can see that unique visits and first visits have a straight upward line, that means that first visits are increasing as the unique visits increase.

viii. Visualising Trend Line between Unique Visits and First Visit:

Since First Visit and Unique Visit have positive correlation so we'll plot the regression line (trend line) between the two and check for any outliers between them.

```
In [18]: px.scatter(
             df, x='FirstVisits', y='UniqueVisits',opacity=0.4,
             trendline='ols', trendline_color_override='purple',title="Regression line for unique visits and first visits"
         )
```

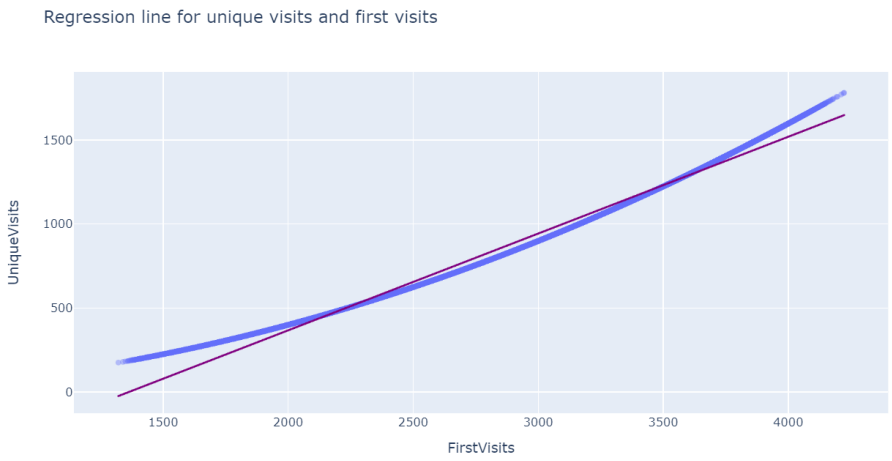


Figure 17: Trend Line between Unique Visit and First Visit

By the above graph we can clearly see that there are no outliers present and its good to go

ix. Visualising Day of Week Distribution:

Now we'll analyze day of week by plotting its pie chart.

```
In [32]: # Pie chart for DayOfWeek distribution
         day_counts = df['DayOfWeek'].value_counts()
         labels = day_counts.index
         sizes = day_counts.values
         colors = ['gold', 'lightcoral', 'lightskyblue', 'lightgreen', 'lightpink', 'lightgray', 'lightyellow']

         plt.figure(figsize=(8, 5))
         plt.pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%', startangle=140)
         plt.title('DayOfWeek Distribution')
         plt.axis('equal')
         plt.show()
```

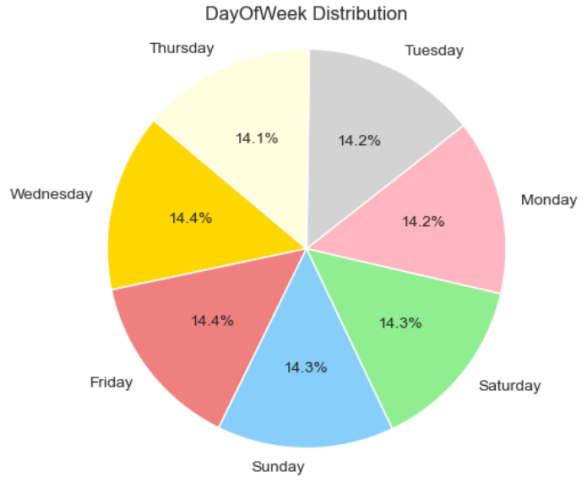


Figure 18: Pie chat showing Day of week distribution

x. Visualization of Violin graph to check for outliers:

Further we can check for any outliers present in our features by using violin graph

Plotting Violin Graph to check for outliers

In [19]:
```
# Defining the columns to visualize
columns_to_visualize = ['PageLoad', 'UniqueVisits', 'FirstVisits', 'ReturnVisits', 'PagesPerVisit']

# Creating a loop to generate violin plots for each column
for column in columns_to_visualize:
    plt.figure(figsize=(3, 1))
    sns.violinplot(x=column, data=df, inner="quart")
    plt.title(f'Violin Plot for {column}')
    plt.xlabel(column)
    plt.ylabel('Density')
    plt.show()
```

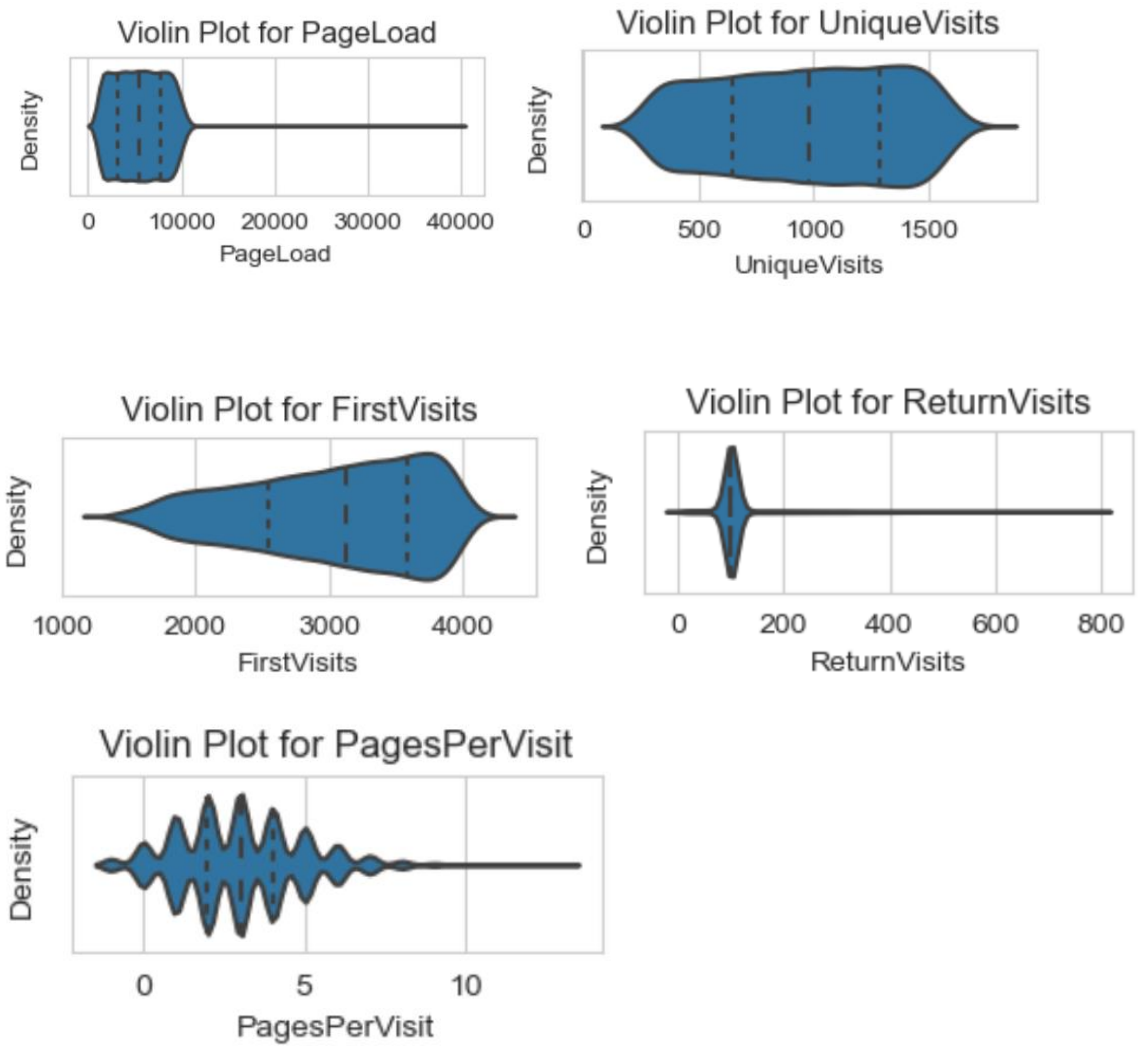


Figure 19: Violin graph to check for outliers.

By the above graphs it can be identified that all are tightly packed and there are no outliers to deal with except PagesPerVisit as we can see except PagesPerVisit other features are tightly packed whereas Zig-Zag figure of PagesPerVisits indicates presence of potential outliers. It can be dealt with during feature engineering.

7.3.4 Feature Engineering:

i. Normalizing Numeric Columns:

In [20]:
```
# Normalizing numeric columns
scaler = MinMaxScaler()
num_cols = ['PageLoad', 'UniqueVisits', 'FirstVisits', 'ReturnVisits','PagesPerVisit']
df[num_cols] = scaler.fit_transform(df[num_cols])
df
```

Out[20]:

	Date	PageLoad	UniqueVisits	FirstVisits	ReturnVisits	DayOfWeek	PagesPerVisit
0	01-01-2017 02:00	0.162791	0.632856	0.734919	0.12500	Sunday	0.571429
1	01-01-2017 03:00	0.077536	0.333541	0.463633	0.12500	Sunday	0.357143
2	01-01-2017 08:00	0.232584	0.856254	0.902447	0.12500	Sunday	0.142857
3	01-01-2017 09:00	0.088142	0.372744	0.503964	0.12500	Sunday	0.214286
4	01-01-2017 10:00	0.060363	0.266957	0.392279	0.42298	Sunday	0.214286
...	...	...	...	...	...	...	...
34598	30-12-2022 13:00	0.127396	0.512757	0.634264	0.12500	Friday	0.142857
34599	30-12-2022 16:00	0.049246	0.222775	0.340572	0.12500	Friday	0.357143
34600	30-12-2022 17:00	0.186915	0.711886	0.796622	0.12500	Friday	0.357143
34601	30-12-2022 18:00	0.217940	0.810828	0.869700	0.12500	Friday	0.357143
34602	30-12-2022 23:00	0.068515	0.298693	0.427094	0.30303	Friday	0.285714

34603 rows × 7 columns

Figure 20: Standardizing Numerical Column.

In our time series data, regression analysis is crucial for forecasting based on historical patterns. To ensure accurate model training, we must standardize the dataset. Standardization equalizes feature scales, preventing bias during training, and ensuring all features contribute effectively to predictions.

ii. Dropping PagesPerVisit Column:

In [21]:
```
#dropping PagesPerVisit because of outliers present in it
df.drop('PagesPerVisit',axis=1,inplace=True)
```

Figure 21: Dropping PagesPerVisit Column

In the above , I have dropped PagesPerVisit, since during our visualisation we found that this column has got potential outliers.

iii. Modifying DayOfWeek:

```
In [22]: #These days have the highest traffic we'll keep them 1 and rest days 0 for better model prediction accuracy
df['days_f']=np.where((df['DayOfWeek']=='Monday') |
                      (df['DayOfWeek']=='Wednesday') |
                      (df['DayOfWeek']=='Friday') |
                      (df['DayOfWeek']=='Saturday'),1,0)

df
```

Out[22]:

	Date	PageLoad	UniqueVisits	FirstVisits	ReturnVisits	DayOfWeek	days_f
0	01-01-2017 02:00	0.162791	0.632856	0.734919	0.12500	Sunday	0
1	01-01-2017 03:00	0.077536	0.333541	0.463633	0.12500	Sunday	0
2	01-01-2017 08:00	0.232584	0.856254	0.902447	0.12500	Sunday	0
3	01-01-2017 09:00	0.088142	0.372744	0.503964	0.12500	Sunday	0
4	01-01-2017 10:00	0.060363	0.266957	0.392279	0.42298	Sunday	0
...	...	...	...	...	...	...	...
34598	30-12-2022 13:00	0.127396	0.512757	0.634264	0.12500	Friday	1
34599	30-12-2022 16:00	0.049246	0.222775	0.340572	0.12500	Friday	1
34600	30-12-2022 17:00	0.186915	0.711886	0.796622	0.12500	Friday	1
34601	30-12-2022 18:00	0.217940	0.810828	0.869700	0.12500	Friday	1
34602	30-12-2022 23:00	0.068515	0.298693	0.427094	0.30303	Friday	1

34603 rows × 7 columns

Figure 22: Dropping PagesPerVisit Column

In the above step, we assigned a value of 1 to the days with the highest traffic, specifically Monday, Wednesday, Friday, and Saturday, which was found during visualization, while assigning a value of 0 to the other days and created a new column name **'days_f'**. This strategic approach enhances the accuracy of our model's predictions. By using this binary representation (1 and 0), we emphasize the importance of high-traffic days (1) in the model's learning process, ensuring it gives more weight to these critical days while appropriately handling the less busy days (0). This approach optimizes the model's ability to capture and forecast traffic patterns effectively.

iv. Setting date as index and dropping day of week

In [23]:
```
#Setting Date as index
df = df.set_index('Date')
#Dropping DayOfWeek
df.drop('DayOfWeek',axis=1,inplace=True)
df
```

Out[23]:

Date	PageLoad	UniqueVisits	FirstVisits	ReturnVisits	days_f
01-01-2017 02:00	0.162791	0.632856	0.734919	0.12500	0
01-01-2017 03:00	0.077536	0.333541	0.463633	0.12500	0
01-01-2017 08:00	0.232584	0.856254	0.902447	0.12500	0
01-01-2017 09:00	0.088142	0.372744	0.503964	0.12500	0
01-01-2017 10:00	0.060363	0.266957	0.392279	0.42298	0
...	...	...	...	...	...
30-12-2022 13:00	0.127396	0.512757	0.634264	0.12500	1
30-12-2022 16:00	0.049246	0.222775	0.340572	0.12500	1
30-12-2022 17:00	0.186915	0.711886	0.796622	0.12500	1
30-12-2022 18:00	0.217940	0.810828	0.869700	0.12500	1
30-12-2022 23:00	0.068515	0.298693	0.427094	0.30303	1

34603 rows × 5 columns

Figure 23: Setting Up date as index and dropping dayofweek

In the previous step, we engineered a new column called 'days_f' based on the 'Day of Week' column. This new column effectively encodes the information previously present in both the 'Date' and 'Day of Week' columns. As a result, we can now safely drop the 'Date' and 'Day of Week' columns from our dataset, as they are no longer necessary for our analysis. This streamlines our data and removes redundant information, simplifying our dataset for further processing.

Now our dataset is ready to go for model prediction.

7.3.5 Model Building

Building the model

In [24]:
```
def print_result(model_name, mse_train, mae_train, mse_test, mae_test):
    print(f"=== Model: {model_name} ===")

    print("Training Results:")
    print(f"Mean Squared Error (MSE) on Training Data: {mse_train:.6f}")
    print(f"Mean Absolute Error (MAE) on Training Data: {mae_train:.6f}")
    print("\n" + "=" * 40 + "\n")

    print("Testing Results:")
    print(f"Mean Squared Error (MSE) on Testing Data: {mse_test:.6f}")
    print(f"Mean Absolute Error (MAE) on Testing Data: {mae_test:.6f}")
    print("\n" + "=" * 40 + "\n")
```

Figure 24: Building the model for Data analysis

```
In [25]: # Define the features and target variable
         X = df.drop(['UniqueVisits'], axis=1).values
         Y = df['UniqueVisits'].values

In [26]: # Split the data into train and test sets
         X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3, random_state=0)

In [27]: # Normalize the data using Min-Max scaling
         scaler = MinMaxScaler()
         X_train = scaler.fit_transform(X_train)
         X_test = scaler.transform(X_test)
```

Figure 25: Splitting the data into training and testing and normalizing the data

In above figures the data is divided into training and testing sets, with 70% of data allocated for training and 30% for testing. Min-Max scaling is applied to standardize features. Model selection, instantiation, training, and evaluation using metrics - MSE and MAE score metrics are used to assess predictive accuracy. The **'print_result 'function** summarizes performance metrics, offering insights into model performance on training and testing data.

i. Long-Short term memory network model:

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) designed for sequential data. LSTMs are particularly effective for time series forecasting due to their ability to capture long-term dependencies in the data. **(Perspectives, 2021)**

Algorithm Steps:

1. **Data Reshaping:** The input data is reshaped into a suitable format for LSTM. In our case, it's reshaped to a 3D tensor. This reshaping is done using the **reshape** method, ensuring that the data is suitable for time series analysis.
2. **Model Architecture:** The LSTM model is built with specified parameters, consisting of the quantity of LSTM units and activation mechanisms.
3. **Compilation:** The model is compiled with an optimizer (here., Adam), and the loss function (here, Mean Squared Error) is defined.
4. **Training:** The LSTM model is trained on the training dataset, iterating over 10 epochs.
5. **Prediction:** After training, Both the training and test datasets are utilized to generate predictions using the model.
6. **Error Calculation:** Mean Squared Error (MSE) and Mean Absolute Error (MAE) are calculated to assess model performance.
7. **Visualization:** Actual vs. Predicted values are visualized using a line plot to evaluate model accuracy.

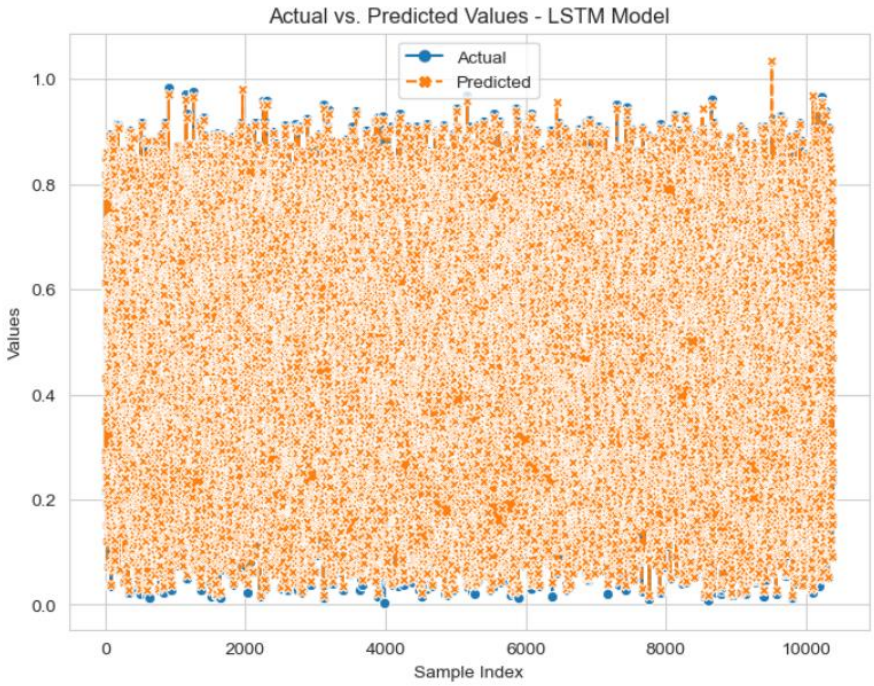
Libraries Used:

- TensorFlow and Keras for LSTM model construction.
- Matplotlib for visualization.
- Scikit Learn for error metric calculations.

LSTM Model

In [61]:
```
# Reshaping the input data for LSTM
X_train_lstm = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
X_test_lstm = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)

# Build the LSTM model
lstm_model = Sequential()
lstm_model.add(LSTM(50, activation='relu', input_shape=(X_train_lstm.shape[1], 1)))
lstm_model.add(Dense(1))  # Output layer with 1 neuron for regression
lstm_model.compile(optimizer='adam', loss='mean_squared_error')

# Train the LSTM model
lstm_model.fit(X_train_lstm, y_train, epochs=10, batch_size=32)

# Making Prediction for LSTM model
lstm_train_predictions = lstm_model.predict(X_train_lstm)
lstm_test_predictions = lstm_model.predict(X_test_lstm)

#Calculating MSE for LSTM model
mse_lstm_train = mean_squared_error(y_train, lstm_train_predictions)
mse_lstm_test = mean_squared_error(y_test, lstm_test_predictions)

#Calculating MAE for LSTM model
mae_lstm_train = mean_absolute_error(y_train, lstm_train_predictions)
mae_lstm_test = mean_absolute_error(y_test, lstm_test_predictions)

# Creating a DataFrame to hold the actual and predicted values
df_lstm_results = pd.DataFrame({'Actual': y_test, 'Predicted': lstm_test_predictions.flatten()})
plt.figure(figsize=(8, 6))
sns.lineplot(data=df_lstm_results, markers=True)
plt.title('Actual vs. Predicted Values - LSTM Model')
plt.grid(True)
plt.show()

print_result("LSTM", mse_lstm_train, mae_lstm_train, mse_lstm_test, mae_lstm_test)
```

```
Epoch 1/10
757/757 [==============================] - 6s 5ms/step - loss: 0.0261
Epoch 2/10
757/757 [==============================] - 3s 4ms/step - loss: 0.0016
Epoch 3/10
757/757 [==============================] - 4s 5ms/step - loss: 4.9445e-04
Epoch 4/10
757/757 [==============================] - 4s 5ms/step - loss: 2.9945e-04
Epoch 5/10
757/757 [==============================] - 4s 5ms/step - loss: 2.1081e-04
Epoch 6/10
757/757 [==============================] - 3s 4ms/step - loss: 1.5363e-04
Epoch 7/10
757/757 [==============================] - 4s 6ms/step - loss: 1.0939e-04
Epoch 8/10
757/757 [==============================] - 4s 5ms/step - loss: 8.0409e-05
Epoch 9/10
757/757 [==============================] - 4s 6ms/step - loss: 6.7265e-05
Epoch 10/10
757/757 [==============================] - 4s 5ms/step - loss: 5.8776e-05
757/757 [==============================] - 3s 3ms/step
325/325 [==============================] - 1s 2ms/step
```




```
=== Model: LSTM ===
Training Results:
Mean Squared Error (MSE) on Training Data: 0.000054
Mean Absolute Error (MAE) on Training Data: 0.001038

========================================

Testing Results:
Mean Squared Error (MSE) on Testing Data: 0.000065
Mean Absolute Error (MAE) on Testing Data: 0.001124

========================================
```

Figure 26: Implementation of LSTM Model with Actual vs Prediction traffic graph and metric report

Here we can see that with LSTM model the MSE on training data is 0.00054 and on testing data is 0.001038 and MAE on Training data is 0.00065 and on testing data is 0.001124.

In summary, these results demonstrate that the LSTM model has been effectively trained and can make accurate predictions, as evidenced by the low MSE and MAE values on both the training and testing datasets.

This shows that the model generalizes to new data effectively, making it a dependable tool for predicting website traffic.

ii. Linear Regression Model:

Linear Regression is a machine learning approach that uses one or more input features to predict a target variable. It presupposes that the input characteristics and the target variable have a linear relationship. **(Python, 2023)**

Linear Regression Workflow:

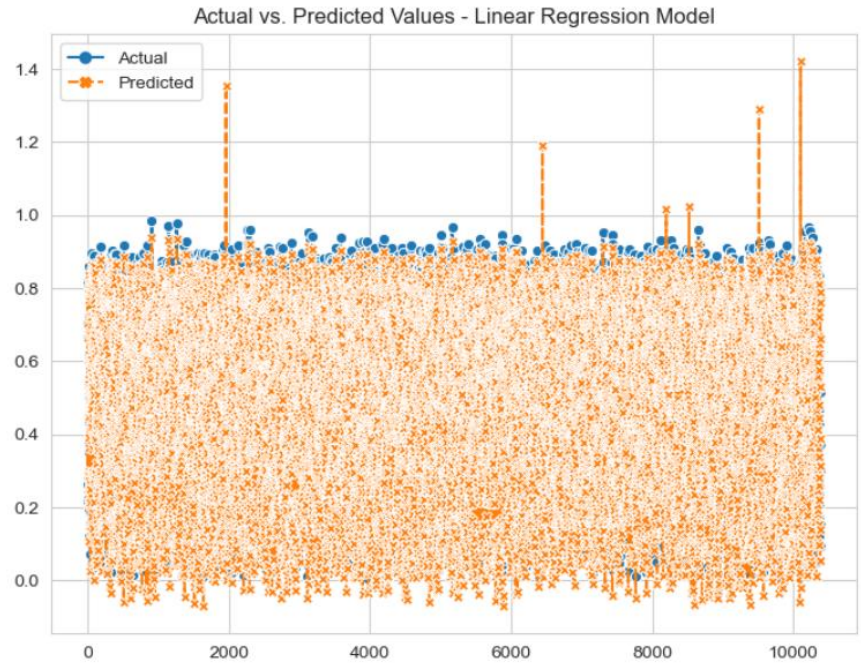
- First, we'll Organize the dataset into input features (X) and the target variable (Y).
- Set up the Linear Regression model and the we'll fit the model.
- Train the model to find the best-fitting linear equation.
- Use the trained model for making predictions.

- Assess model accuracy using metrics - MSE and MAE.
- Create visual comparisons of actual vs. predicted values for better understanding.

It's a methodical approach that aims to uncover relationships within data and make informed predictions based on those relationships.

Linear Regression Model

```
In [62]: # Building and training the Linear Regression model
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)

#Making Prediction for Linear Regression Model
linear_train_predictions = linear_model.predict(X_train)
linear_test_predictions = linear_model.predict(X_test)

#Calculating MSE for Linear Regression Model
mse_linear_train = mean_squared_error(y_train, linear_train_predictions)
mse_linear_test = mean_squared_error(y_test, linear_test_predictions)

#Calculating MAE for Linear Regression Model
mae_linear_train = mean_absolute_error(y_train, linear_train_predictions)
mae_linear_test = mean_absolute_error(y_test, linear_test_predictions)

# Creating a DataFrame to hold the actual and predicted values
df_linear_results = pd.DataFrame({'Actual': y_test, 'Predicted': linear_test_predictions.flatten()})
plt.figure(figsize=(8, 6))
sns.lineplot(data=df_linear_results, markers=True)
plt.title('Actual vs. Predicted Values - Linear Regression Model')
plt.grid(True)
plt.show()

print_result("Linear Regression", mse_linear_train, mae_linear_train, mse_linear_test, mae_linear_test)
```




```
=== Model: Linear Regression ===
Training Results:
Mean Squared Error (MSE) on Training Data: 0.000397
Mean Absolute Error (MAE) on Training Data: 0.013064

========================================

Testing Results:
Mean Squared Error (MSE) on Testing Data: 0.000453
Mean Absolute Error (MAE) on Testing Data: 0.013229

========================================
```

Figure 27: Implementation of Linear Regression Model with Actual vs Prediction traffic graph and metric report

Training Results:

MSE: 0.000397, indicating relatively low errors on the training data.

MAE: 0.013064, signifying small absolute errors on the training data.

Testing Results:

MSE: 0.000453, reflecting reasonable performance on new, unseen data.

MAE: 0.013229, demonstrating consistent and acceptable accuracy on the testing data.

In both cases, lower values of Mean Squared Error (MSE) and Mean Absolute Error (MAE) are highly desirable. These low error values indicate the model's exceptional ability to make accurate predictions, both during training on known data and when applied to new, unseen data.

iii. K-Nearest Neighbor Regressor Model:

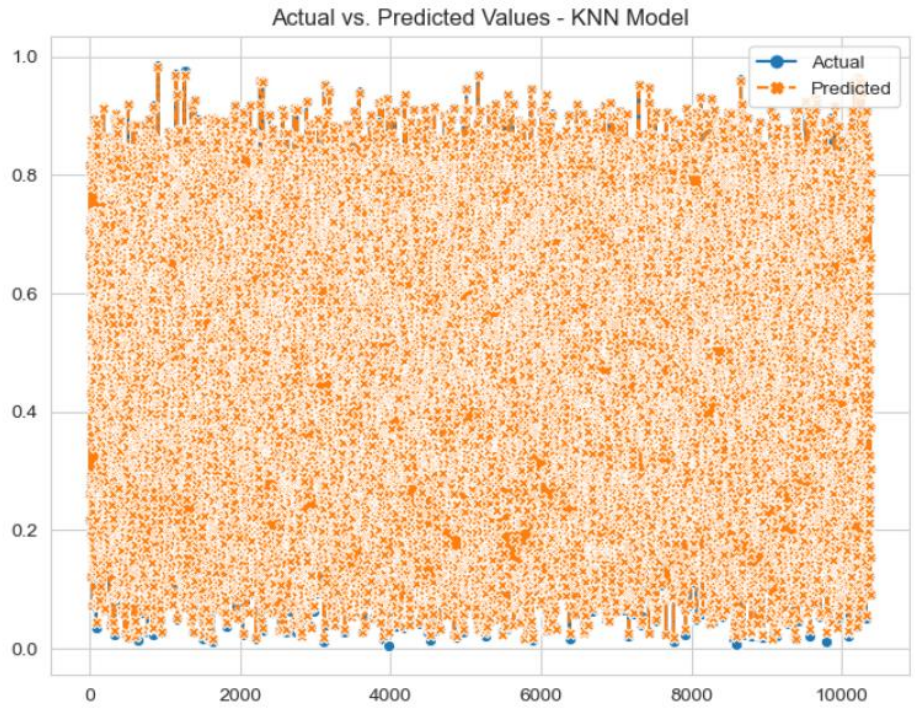
KNN, or K-Nearest Neighbors A supervised machine learning approach called regression is utilized for regression tasks. The training dataset's 'k' nearest data points are taken into account for predicting the target variable. **(Teixeira-Pinto, 2022)**

KNN workflow-

- We initialize the KNN model with 5 neighbors (n_neighbors=5).
- The model is trained on the training dataset, and predictions are made on both training and test data.
- We calculate the Mean Squared Error (MSE) and Mean Absolute Error (MAE) to evaluate the model's performance.
- Actual vs. Predicted values are visualized using a line plot for model assessment.

K-Nearest Neighbour Model

In [64]:
```
# Building and training the KNN model
knn_model = KNeighborsRegressor(n_neighbors=5)
knn_model.fit(X_train, y_train)

#Making Prediction for KNN Model
knn_train_predictions = knn_model.predict(X_train)
knn_test_predictions = knn_model.predict(X_test)

#Calculating MSE for KNN Model
mse_knn_train = mean_squared_error(y_train, knn_train_predictions)
mse_knn_test = mean_squared_error(y_test, knn_test_predictions)

#Calculating MAE for KNN Model
mae_knn_train = mean_absolute_error(y_train, knn_train_predictions)
mae_knn_test = mean_absolute_error(y_test, knn_test_predictions)

# Creating a DataFrame to hold the actual and predicted values
df_knn_results = pd.DataFrame({'Actual': y_test, 'Predicted': knn_test_predictions.flatten()})
plt.figure(figsize=(8, 6))
sns.lineplot(data=df_knn_results, markers=True)
plt.title('Actual vs. Predicted Values - KNN Model')
plt.grid(True)
plt.show()

print_result("KNN Regression", mse_knn_train, mae_knn_train, mse_knn_test, mae_knn_test)
```




```
=== Model: KNN Regression ===
Training Results:
Mean Squared Error (MSE) on Training Data: 0.000007
Mean Absolute Error (MAE) on Training Data: 0.000498

========================================

Testing Results:
Mean Squared Error (MSE) on Testing Data: 0.000009
Mean Absolute Error (MAE) on Testing Data: 0.000629

========================================
```

Figure 28: Implementation of KNN Regressor Model with Actual vs Prediction traffic graph and metric report

Training Results:

MSE: 0.000007 (Indicating very low errors on the training data)

MAE: 0.000498 (Signifying very small absolute errors on the training data)

Testing Results:

MSE: 0.000009 (Reflecting extremely low errors on new, unseen data)

MAE: 0.000629 (Demonstrating highly accurate predictions on the testing data)

In both cases, lower values are better. The Mean Squared Error (MSE) measures the average squared difference between predicted and actual values, while the Mean Absolute Error (MAE) calculates the average absolute difference. These low MSE and MAE values indicate that the KNN Regression model makes highly accurate predictions, with minimal errors on both the training and testing datasets.

iv. Support Vector Regression Model

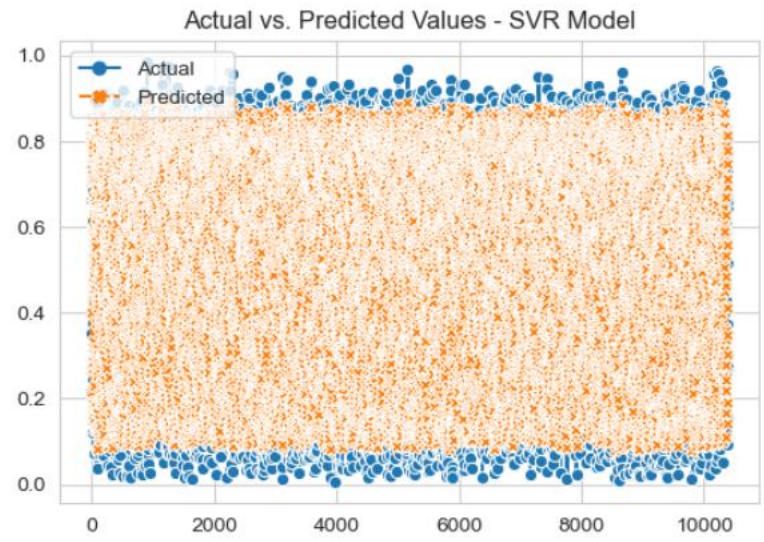
Support Vector Regression (SVR) is a supervised machine learning algorithm used for regression tasks. It aims to predict the target variable by creating a hyperplane that best fits the data while minimizing errors. SVR is particularly effective in handling non-linear relationships in data.

SVR Algorithm Overview:

- The SVR model is initialized with a radial basis function kernel ('rbf') and specific hyperparameters like C and epsilon.(Here C=1.0 and epsilon=0.1)
- Training is performed on the training dataset, and predictions are generated for both training and test data.
- Mean Squared Error (MSE) and Mean Absolute Error (MAE) are calculated to assess the model's accuracy.
- Actual vs. Predicted values are visualized using a line plot, facilitating model evaluation and comparison.

Support Vector Regression Model

In [69]:
```
# Building and training the SVR model
svr_model = SVR(kernel='rbf', C=1.0, epsilon=0.1)
svr_model.fit(X_train, y_train)

#Making Prediction for SVR Model
svr_train_predictions = svr_model.predict(X_train)
svr_test_predictions = svr_model.predict(X_test)

#Calculating MSE for SVR Model
mse_svr_train = mean_squared_error(y_train, svr_train_predictions)
mse_svr_test = mean_squared_error(y_test, svr_test_predictions)

#Calculating MAE for SVR Model
mae_svr_train = mean_absolute_error(y_train, svr_train_predictions)
mae_svr_test = mean_absolute_error(y_test, svr_test_predictions)

# Creating a DataFrame to hold the actual and predicted values
df_svr_results = pd.DataFrame({'Actual': y_test, 'Predicted':  svr_test_predictions.flatten()})
plt.figure(figsize=(6, 4))
sns.lineplot(data=df_svr_results, markers=True)
plt.title('Actual vs. Predicted Values - SVR Model')
plt.grid(True)
plt.show()

print_result("SVR", mse_svr_train, mae_svr_train, mse_svr_test, mae_svr_test)
```




```
=== Model: SVR ===
Training Results:
Mean Squared Error (MSE) on Training Data: 0.001660
Mean Absolute Error (MAE) on Training Data: 0.036915

========================================

Testing Results:
Mean Squared Error (MSE) on Testing Data: 0.001689
Mean Absolute Error (MAE) on Testing Data: 0.037232

========================================
```

Figure 29: Implementation of SVR Model with Actual vs Prediction traffic graph and metric report

Training Results:

MSE: 0.001660 (Indicating very low errors on the training data)

MAE: 0.036915 (Signifying very small absolute errors on the training data)

Testing Results:

MSE: 0.001689 (Reflecting extremely low errors on new, unseen data)

MAE: 0.037232 (Demonstrating highly accurate predictions on the testing data)

both the training and testing results suggest that the SVR model is capable of making predictions with reasonable accuracy, as indicated by the low MSE and MAE values. However, the specific interpretation of "reasonable" accuracy may depend on the context of your application and the scale of the target variable. Lower MSE and MAE values generally imply better predictive performance.

v. Neural Network Model (MLP Regressor):

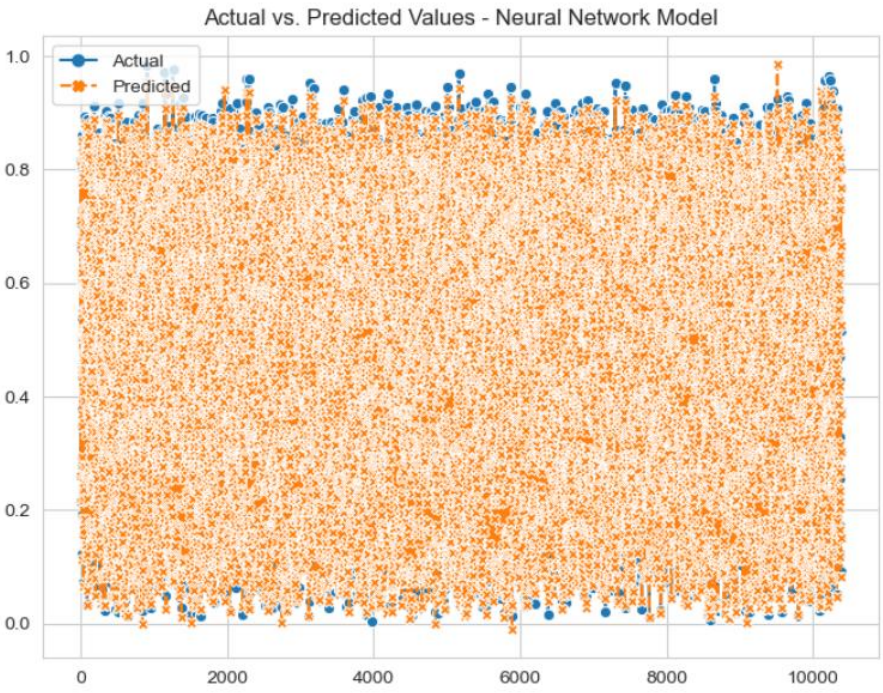
The MLPRegressor (Multi-layer Perceptron Regressor) is a type of artificial neural network used for regression tasks. It's a versatile and powerful model capable of learning complex relationships between input features and the target variable.

NN workflow:

- We initialize the MLPRegressor model with three hidden layers, each containing eight neurons, and use the Rectified Linear Unit (ReLU) activation function.
- The 'adam' solver is employed for optimization, with a maximum of 500 iterations, and a random seed is set for reproducibility.
- The model is trained on the training dataset, and predictions are made on both training and test data.
- We calculate the Mean Squared Error (MSE) and Mean Absolute Error (MAE) to evaluate the model's performance.

Neural Network MLPRegressor Model

In [70]:
```
# Building and compile the Neural Network MLPRegressor model
mlp_model = MLPRegressor(hidden_layer_sizes=(8,8,8), activation='relu', solver='adam', max_iter=500, random_state=0)
mlp_model.fit(X_train, y_train)

#Making Prediction for MLP Model
mlp_train_predictions = mlp_model.predict(X_train)
mlp_test_predictions = mlp_model.predict(X_test)

#Calculating MSE for MLP Model
mse_mlp_train = mean_squared_error(y_train, mlp_train_predictions)
mse_mlp_test = mean_squared_error(y_test, mlp_test_predictions)

#Calculating MAE for MLP Model
mae_mlp_train = mean_absolute_error(y_train, mlp_train_predictions)
mae_mlp_test = mean_absolute_error(y_test, mlp_test_predictions)

# Creating a DataFrame to hold the actual and predicted values
df_nml_results = pd.DataFrame({'Actual': y_test, 'Predicted': mlp_test_predictions.flatten()})
plt.figure(figsize=(8, 6))
sns.lineplot(data=df_nml_results, markers=True)
plt.title('Actual vs. Predicted Values - Neural Network Model')
plt.grid(True)
plt.show()

print_result("MLPRegressor", mse_mlp_train, mae_mlp_train, mse_mlp_test, mae_mlp_test)
```




```
=== Model: MLPRegressor ===
Training Results:
Mean Squared Error (MSE) on Training Data: 0.000063
Mean Absolute Error (MAE) on Training Data: 0.005733

========================================

Testing Results:
Mean Squared Error (MSE) on Testing Data: 0.000067
Mean Absolute Error (MAE) on Testing Data: 0.005723

========================================
```

Figure 30: Implementation of Neural Network Model (MLP Regressor) with Actual vs Prediction traffic graph and metric report

Here we can see that with NN MLP model the MSE on training data is 0.000063 and on testing data is 0.005733 and MAE on Training data is 0.000067 and on testing data is 0.005723.

These results suggest that the MLPRegressor model is performing well in both training and testing scenarios. It exhibits low prediction errors, indicating its ability to accurately predict the target variable. The small MAE values also imply that the model's predictions are quite precise and reliable.

7.3.6 Result and discussion before Optimization:

Since Each Model Showed Low metric score it's hard to determine the optimal model performance, so to get clearer view towards model's performance, grouped bar graph is plotted.

Comparison Between Each Model According to Training and Testing MSE

In [71]:
```
# Update the models list
models = ['LSTM', 'Linear Regression', 'KNN Regression', 'MLPRegressor', 'SVR']

# Add MSE values for all models to the lists
train_mse = [mse_lstm_train, mse_linear_train, mse_knn_train, mse_mlp_train, mse_svr_train]
test_mse = [mse_lstm_test, mse_linear_test, mse_knn_test, mse_mlp_test, mse_svr_test]

# Create a grouped bar chart to compare the MSE of each model according to training and testing
plt.figure(figsize=(12, 6))
bar_width = 0.35
index = np.arange(len(models))
plt.bar(index - bar_width/2, train_mse, bar_width, label='Training MSE', alpha=0.7)
plt.bar(index + bar_width/2, test_mse, bar_width, label='Testing MSE', alpha=0.7)
plt.xlabel('Models')
plt.ylabel('Mean Squared Error (MSE)')
plt.xticks(index, models)
plt.legend()
plt.title('Model Comparison (Training vs. Testing MSE)')
plt.show()
```

Figure 31: Group bar graph to show model comparison of MSE results on training and testing results.

Comparison Between Each Model According to Training and Testing MAE

In [72]:
```
# Update the models list
models = ['LSTM', 'Linear Regression', 'KNN Regression', 'MLPRegressor', 'SVR']

# Add MAE values for all models to the lists
train_mae = [mae_lstm_train, mae_linear_train, mae_knn_train, mae_mlp_train, mae_svr_train]
test_mae = [mae_lstm_test, mae_linear_test, mae_knn_test, mae_mlp_test, mae_svr_test]

# Create a grouped bar chart to compare the MAE of each model according to training and testing
plt.figure(figsize=(12, 6))
bar_width = 0.35
index = np.arange(len(models))
plt.bar(index - bar_width/2, train_mae, bar_width, label='Training MAE', alpha=0.7)
plt.bar(index + bar_width/2, test_mae, bar_width, label='Testing MAE', alpha=0.7)
plt.xlabel('Models')
plt.ylabel('Mean Absolute Error (MAE)')
plt.xticks(index, models)
plt.legend()
plt.title('Model Comparison (Training vs. Testing MAE)')
plt.show()
```

Figure 32: Group bar graph to show model comparison of MSE results on training and testing results.

The grouped bar graphs provide a clear performance comparison among the different regression models. It is evident that the KNN Regressor stands out as the top-performing model, closely followed by LSTM, Neural Network, Linear Regression, and SVR.

KNN Regressor demonstrates superior predictive accuracy with the lowest MSE and MAE scores, indicating its effectiveness in capturing the underlying patterns in the dataset.

On the other hand, SVR emerges as the least suitable model for this specific dataset. This conclusion is supported by its notably higher MSE and MAE values compared to the other models, signifying a less accurate fit to the data.

In summary, the visualization effectively highlights the varying performance levels of these regression models, making it evident that KNN Regressor is the most promising choice for this prediction task, Followed by LSTM, while SVR should be reconsidered due to its comparatively poorer performance.

7.3.7 Hyperparameter Tuning:

Hyperparameter tuning is the crucial process of optimizing machine learning models by adjusting the algorithms' settings and behaviors. It involves tweaking hyperparameters that control the model training process in order to improve performance and generalization capabilities. By fine-tuning hyperparameters, the model's ability to generalize from the training data to new, unseen data can be enhanced. Grid search, random search, Bayesian optimization and other techniques are employed to methodically explore combinations of hyperparameter values and select those that improve accuracy, generalization or other desired metrics. The overarching goal of hyperparameter tuning is striking an ideal balance between overfitting and underfitting. This calibration yields an optimized model that generalizes well and delivers optimal performance on real-world data beyond just the training examples. Careful tuning adapts the model to the problem at hand and dataset characteristics, rather than using default parameters. The result is more robust models that make accurate predictions on new data.

For Tuning we'll use **GridSearchCV –** It is a technique for hyperparameter tuning that exhaustively searches a predefined grid of hyperparameters using cross-validation to evaluate and select the optimal model configuration. It iterates through the hyperparameter space, training and evaluating the model on different splits of the data to identify the ideal combination of hyperparameters that maximizes model performance.

```
In [77]: from sklearn.model_selection import GridSearchCV
```

Figure 33: Importing GridSearchCV library.

i.Tuning LSTM Model:

Here Tuning in the below parameters to get optimized results.

Batch Sizes: Trying different batch sizes (here,, 32, 64) to control how many examples are used in each training iteration.

Epochs: Testing various epoch values (e.g., 10, 20, 50) to find the right number of times the model iterates over the dataset.

Number of LSTM Units: Experimenting with different unit counts (e.g., 50, 100) to adjust model complexity.

Optimization Algorithm: Evaluating different optimizers (e.g., 'adam', 'rmsprop') to update model weights.

The code explores these options systematically, training and evaluating models with different combinations of these hyperparameters. The goal is to discover the best settings for accurate time series forecasting. **(*The Grid Search Hyper-parameter Sets for the CNN + LSTM Model.*, n.d.)**

Hyperparameter Tuning For LSTM

In [108]:
```
# Function to create the LSTM model
def create_lstm_model(units=50, optimizer='adam'):
    model = Sequential()
    model.add(LSTM(units, activation='relu', input_shape=(X_train_lstm.shape[1], 1)))
    model.add(Dense(1))
    model.compile(optimizer=optimizer, loss='mean_squared_error')
    return model

# Define hyperparameters and their possible values
batch_sizes = [32, 64]
epochs_values = [10,20,50]
units_values = [50, 100]  # Number of LSTM units
optimizer_values = ['adam', 'rmsprop']

best_lstm_mse_train = float('inf')
best_lstm_mse_test = float('inf')
best_lstm_mae_train = float('inf')
best_lstm_mae_test = float('inf')
best_lstm_hyperparameters = None

for batch_size in batch_sizes:
    for epochs in epochs_values:
        for units in units_values:
            for optimizer in optimizer_values:
                # Create and compile the model
                lstm_model = create_lstm_model(units=units, optimizer=optimizer)

                # Train the model
                lstm_model.fit(X_train_lstm, y_train, epochs=epochs, batch_size=batch_size, verbose=0)

                # Make predictions
                lstm_train_predictions = lstm_model.predict(X_train_lstm)
                lstm_test_predictions = lstm_model.predict(X_test_lstm)

                # Calculate MSE and MAE
                mse_train = mean_squared_error(y_train, lstm_train_predictions)
                mse_test = mean_squared_error(y_test, lstm_test_predictions)
                mae_train = mean_absolute_error(y_train, lstm_train_predictions)
                mae_test = mean_absolute_error(y_test, lstm_test_predictions)

                if mse_test < best_lstm_mse_test:
                    best_lstm_mse_train = mse_train
                    best_lstm_mse_test = mse_test
                    best_lstm_mae_train = mae_train
                    best_lstm_mae_test = mae_test
                    best_lstm_hyperparameters = {
                        'batch_size': batch_size,
                        'epochs': epochs,
                        'units': units,
                        'optimizer': optimizer
                    }

print("Best Hyperparameters for LSTM:", best_lstm_hyperparameters)
print("MSE for LSTM (Training):", best_lstm_mse_train)
print("MSE for LSTM (Testing):", best_lstm_mse_test)
print("MAE for LSTM (Training):", best_lstm_mae_train)
print("MAE for LSTM (Testing):", best_lstm_mae_test)
```

```
757/757 [==============================] - 2s 3ms/step
325/325 [==============================] - 1s 3ms/step
757/757 [==============================] - 3s 4ms/step
325/325 [==============================] - 1s 4ms/step
757/757 [==============================] - 3s 4ms/step
325/325 [==============================] - 1s 4ms/step
757/757 [==============================] - 2s 2ms/step
325/325 [==============================] - 1s 2ms/step
757/757 [==============================] - 2s 3ms/step
325/325 [==============================] - 1s 3ms/step
757/757 [==============================] - 3s 4ms/step
325/325 [==============================] - 1s 4ms/step
757/757 [==============================] - 3s 4ms/step
325/325 [==============================] - 1s 4ms/step
Best Hyperparameters for LSTM: {'batch_size': 64, 'epochs': 50, 'units': 100, 'optimizer': 'adam'}
MSE for LSTM (Training): 6.269510361388097e-07
MSE for LSTM (Testing): 6.363231503434297e-07
MAE for LSTM (Training): 0.0006646094357559 04
MAE for LSTM (Testing): 0.0006687879632950571
```

Figure 34: Implementing Hyperparameter Tuning on LSTM model and

ii. Tuning Linear Regression Model:

To Tune Linear Regression model, we use Ridge Regression

Alpha Values: We define a range of alpha values (e.g., 0.001, 0.01, 0.1, 1, 10) that control the regularization strength. Ridge Regression uses L2 regularization to prevent overfitting.

Grid Search: We perform a grid search with cross-validation (cv=5) to find the best alpha value. The search aims to minimize the negative mean squared error (neg_mean_squared_error). negating MSE because we effectively turn it into a "higher is better" metric for optimization purposes.

Best Alpha: The best alpha value is determined by grid search (e.g., best_ridge_alpha = 0.001).

Model Evaluation: We calculate and report the Mean Squared Error (MSE) and Mean Absolute Error (MAE) for both training and testing data. These metrics assess the model's performance.

The code systematically explores different alpha values to optimize Ridge Regression, helping to find the best regularization strength for accurate predictions**. (Brownlee, 2020)**

Hyperparameter Tuning for Linear Regression

In [90]:
```
from sklearn.linear_model import Ridge

# Define a range of alpha values to tune
param_grid = {'alpha': [0.001, 0.01, 0.1, 1, 10]}

ridge_model = Ridge()
grid_search = GridSearchCV(ridge_model, param_grid, cv=5, scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train)

best_ridge_alpha = grid_search.best_params_['alpha']
best_linear_mse_train = -grid_search.best_score_
best_linear_mse_test = mean_squared_error(y_test, grid_search.predict(X_test))
best_linear_mae_train = mean_absolute_error(y_train, grid_search.predict(X_train))
best_linear_mae_test = mean_absolute_error(y_test, grid_search.predict(X_test))

print("Best Hyperparameter (Alpha) for Linear Regression:", best_ridge_alpha)
print("MSE for Linear Regression (Training):", best_linear_mse_train)
print("MSE for Linear Regression (Testing):", best_linear_mse_test)
print("MAE for Linear Regression (Training):", best_linear_mae_train)
print("MAE for Linear Regression (Testing):", best_linear_mae_test)
```

```
Best Hyperparameter (Alpha) for Linear Regression: 1
MSE for Linear Regression (Training): 0.0004117516682994235
MSE for Linear Regression (Testing): 0.00044555047565838986
MAE for Linear Regression (Training): 0.013733439720793739
MAE for Linear Regression (Testing): 0.01388322963331719
```

Figure 35: Implementing Hyperparameter Tuning on Linear Regression model to get Best Params and Metric report

iii. Tuning KNN Model:

To Tune the KNN model we define a parameter grid param_grid that includes a range of values for the hyperparameter 'n_neighbors,' representing the number of nearest neighbors to consider (e.g., [3, 5, 7, 9, 11]).

Model Initialization: We create an instance of the KNeighborsRegressor model.

Grid Search: GridSearchCV is employed to perform a grid search with 5-fold cross-validation (cv=5). The objective is to minimize the negative mean squared error (neg_mean_squared_error) during the search.

Best Hyperparameters: The best hyperparameters are identified based on the grid search results, including the optimal 'n_neighbors' value.

Model Training and Prediction: The KNN Regression model is trained on the training dataset, and predictions are made for both the training and test datasets.

Model Evaluation: We calculate the Mean Squared Error (MSE) and Mean Absolute Error (MAE) for both the training and testing data to evaluate the model's performance.

This code systematically explores different 'n_neighbors' values to optimize the KNN Regression model, ultimately finding the best 'k' value for making accurate predictions on the given dataset.**(Brownlee,2020)**

Hyperparameter Tuning for KNN

In [91]:
```
param_grid = {'n_neighbors': [3, 5, 7, 9, 11]}

knn_model = KNeighborsRegressor()
grid_search = GridSearchCV(knn_model, param_grid, cv=5, scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train)

best_knn_mse_train = -grid_search.best_score_
best_knn_mse_test = mean_squared_error(y_test, grid_search.predict(X_test))
best_knn_mae_train = mean_absolute_error(y_train, grid_search.predict(X_train))
best_knn_mae_test = mean_absolute_error(y_test, grid_search.predict(X_test))
best_knn_hyperparameters = grid_search.best_params_

print("Best Hyperparameters for KNN Regression:", best_knn_hyperparameters)
print("MSE for KNN Regression (Training):", best_knn_mse_train)
print("MSE for KNN Regression (Testing):", best_knn_mse_test)
print("MAE for KNN Regression (Training):", best_knn_mae_train)
print("MAE for KNN Regression (Testing):", best_knn_mae_test)
```

```
Best Hyperparameters for KNN Regression: {'n_neighbors': 9}
MSE for KNN Regression (Training): 1.0021678986155313e-05
MSE for KNN Regression (Testing): 1.041774469171505e-05
MAE for KNN Regression (Training): 0.0005993558180940053
MAE for KNN Regression (Testing): 0.0006987792619685491
```

Figure 36: Implementing Hyperparameter Tuning on KNN Regression model to get Best Params and Metric report

iv. Tuning SVR Model

To tune SVR we define a parameter grid param_grid that includes a range of values for two hyperparameters: 'C' (penalty parameter) and 'epsilon' (the epsilon in the epsilon-insensitive loss function). Different combinations of these hyperparameters are considered (e.g., 'C': [0.1, 1, 10] and 'epsilon': [0.01, 0.1, 1]).

Model Initialization: We create an instance of the SVR model with a radial basis function (RBF) kernel.

Grid Search: GridSearchCV is used to perform a grid search with 5-fold cross-validation (cv=5). The objective is to minimize the negative mean squared error (neg_mean_squared_error) during the search.

Best Hyperparameters: The best hyperparameters are determined based on the grid search results, including the optimal 'C' and 'epsilon' values.

Model Training and Prediction: The SVR model is trained on the training dataset, and predictions are made for both the training and test datasets.

Model Evaluation: We calculate the Mean Squared Error (MSE) and Mean Absolute Error (MAE) for both the training and testing data to evaluate the model's performance.

This code systematically explores different combinations of 'C' and 'epsilon' values to optimize the SVR model, helping to find the best hyperparameters for making accurate predictions on the given dataset. **(*SVM Hyperparameter Tuning Using GridSearchCV - Velocity Business Solutions Limited*, 2020)**

Hyperparameter Tuning for SVR

In [92]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error

param_grid = {'C': [0.1, 1, 10],
              'epsilon': [0.01, 0.1, 1]}

svr_model = SVR(kernel='rbf')
grid_search = GridSearchCV(svr_model, param_grid, cv=5, scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train)

best_svr_mse_train = -grid_search.best_score_
best_svr_mse_test = mean_squared_error(y_test, grid_search.predict(X_test))
best_svr_mae_train = mean_absolute_error(y_train, grid_search.predict(X_train))
best_svr_mae_test = mean_absolute_error(y_test, grid_search.predict(X_test))
best_svr_hyperparameters = grid_search.best_params_

print("Best Hyperparameters for SVR:", best_svr_hyperparameters)
print("MSE for SVR (Training):", best_svr_mse_train)
print("MSE for SVR (Testing):", best_svr_mse_test)
print("MAE for SVR (Training):", best_svr_mae_train)
print("MAE for SVR (Testing):", best_svr_mae_test)
```

```
Best Hyperparameters for SVR: {'C': 10, 'epsilon': 0.01}
MSE for SVR (Training): 4.56581118736526e-05
MSE for SVR (Testing): 4.3944370137617304e-05
MAE for SVR (Training): 0.005926097298212769
MAE for SVR (Testing): 0.00590353849902086
```

Figure 37: Implementing Hyperparameter Tuning on SVR model to get Best Params and Metric report

v. Tuning Neural Network MLP Regressor Model

To tune this we define a parameter grid param_grid that includes various hyperparameters to tune the MLPRegressor model. These hyperparameters include 'hidden_layer_sizes' (number of neurons in each hidden layer), 'alpha' (L2 regularization parameter), and 'max_iter' (maximum number of iterations).

Model Initialization: We create an instance of the MLPRegressor model with specified settings for activation function ('relu'), solver ('adam'), and a random seed for reproducibility.

Grid Search: GridSearchCV is employed to perform a grid search with 5-fold cross-validation (cv=5). The objective is to minimize the negative mean squared error (neg_mean_squared_error) during the search.

Best Hyperparameters: The best hyperparameters are determined based on the grid search results, including the optimal 'hidden_layer_sizes,' 'alpha,' and 'max_iter' values.

Model Training and Prediction: The MLPRegressor model is trained on the training dataset, and predictions are made for both the training and test datasets.

Model Evaluation: We calculate the Mean Squared Error (MSE) and Mean Absolute Error (MAE) for both the training and testing data to evaluate the model's performance.

This code systematically explores different combinations of hyperparameters to optimize the MLPRegressor model, helping to find the best hyperparameters for accurate predictions on the given dataset.

Hyperparameter Tuning for MLPRegressor

In [93]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error

param_grid = {'hidden_layer_sizes': [(8, 8, 8), (16, 16, 16)],
              'alpha': [0.0001, 0.001],
              'max_iter': [500, 1000]}

mlp_model = MLPRegressor(activation='relu', solver='adam', random_state=0)
grid_search = GridSearchCV(mlp_model, param_grid, cv=5, scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train)

best_mlp_mse_train = -grid_search.best_score_
best_mlp_mse_test = mean_squared_error(y_test, grid_search.predict(X_test))
best_mlp_mae_train = mean_absolute_error(y_train, grid_search.predict(X_train))
best_mlp_mae_test = mean_absolute_error(y_test, grid_search.predict(X_test))
best_mlp_hyperparameters = grid_search.best_params_

print("Best Hyperparameters for MLPRegressor:", best_mlp_hyperparameters)
print("MSE for MLPRegressor (Training):", best_mlp_mse_train)
print("MSE for MLPRegressor (Testing):", best_mlp_mse_test)
print("MAE for MLPRegressor (Training):", best_mlp_mae_train)
print("MAE for MLPRegressor (Testing):", best_mlp_mae_test)
```

```
Best Hyperparameters for MLPRegressor: {'alpha': 0.001, 'hidden_layer_sizes': (16, 16, 16), 'max_iter': 500}
MSE for MLPRegressor (Training): 2.28165436487103e-05
MSE for MLPRegressor (Testing): 1.9307665210180068e-05
MAE for MLPRegressor (Training): 0.0032151928404909 15
MAE for MLPRegressor (Testing): 0.0032589584241428434
```

Figure 38: Implementing Hyperparameter Tuning on Neural Network MLP Regression model to get Best Params and Metric report

7.3.8 Result

In [110]:
```
# Creating a list of dictionaries, where each dictionary represents a model's best hyperparameters
best_hyperparameters_list = [
    {'Model': 'LSTM', 'Best Parameters': best_lstm_hyperparameters},
    {'Model': 'Ridge Regression', 'Best Parameters': {'Alpha': best_ridge_alpha}},
    {'Model': 'KNN Regression', 'Best Parameters': best_knn_hyperparameters},
    {'Model': 'MLPRegressor', 'Best Parameters': best_mlp_hyperparameters},
    {'Model': 'SVR', 'Best Parameters': best_svr_hyperparameters}

]

# Create a DataFrame from the list of dictionaries
best_hyperparameters_df = pd.DataFrame(best_hyperparameters_list)

# Display the DataFrame as a table
print(best_hyperparameters_df)
```

```
              Model                                    Best Parameters
0              LSTM  {'batch_size': 64, 'epochs': 50, 'units': 100,...
1  Ridge Regression                                        {'Alpha': 1}
2    KNN Regression                                   {'n_neighbors': 9}
3      MLPRegressor  {'alpha': 0.001, 'hidden_layer_sizes': (16, 16...
4               SVR                           {'C': 10, 'epsilon': 0.01}
```

Figure 39: Demonstrating Model and its best parameters after optimization.

In [109]:
```
# Defining the models and their corresponding MSE and MAE values
models = ['LSTM', 'Linear Regression', 'KNN Regression', 'SVR', 'MLPRegressor']
normal_mse = [mse_lstm_test, mse_linear_test, mse_knn_test, mse_svr_test, mse_mlp_test]
best_mse = [best_lstm_mse_test, best_linear_mse_test, best_knn_mse_test, best_svr_mse_test, best_mlp_mse_test]
normal_mae = [mae_lstm_test, mae_linear_test, mae_knn_test, mae_svr_test, mae_mlp_test]
best_mae = [mae_lstm_test, best_linear_mae_test, best_knn_mae_test, best_svr_mae_test, best_mlp_mae_test]

# Creating subplots for MSE and MAE
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(10, 8))

# Plotting MSE values
axes[0].bar(models, normal_mse, width=0.4, label='Normal MSE', align='center', alpha=0.7)
axes[0].bar(models, best_mse, width=0.4, label='Best MSE', align='edge', alpha=0.7)
axes[0].set_ylabel('Mean Squared Error (MSE)')
axes[0].set_title('Comparison of Normal vs. Best MSE')
axes[0].legend()

# Plotting MAE values
axes[1].bar(models, normal_mae, width=0.4, label='Normal MAE', align='center', alpha=0.7)
axes[1].bar(models, best_mae, width=0.4, label='Best MAE', align='edge', alpha=0.7)
axes[1].set_ylabel('Mean Absolute Error (MAE)')
axes[1].set_title('Comparison of Normal vs. Best MAE')
axes[1].legend()

# Adjusting the layout
plt.tight_layout()

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

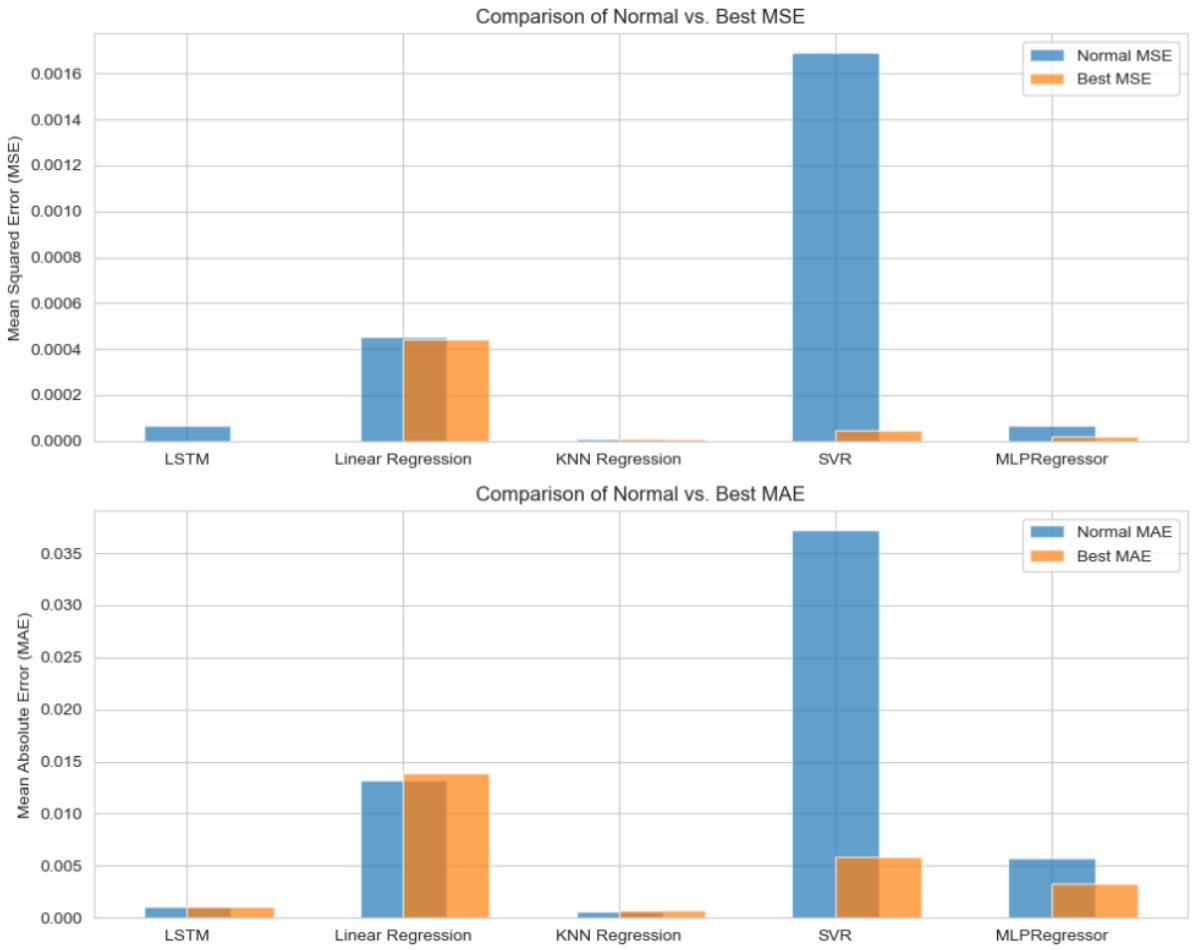


Figure 40: Grouped Bar graph plot for comparison between MSE and MAE scores on training and testing on each model after optimization.

	MSE before optimization		MSE After Optimization		MAE before Optimization		MAE After Optimization	
	Train	Test	Train	Test	Train	Test	Train	Test
LSTM	0.000054	0.000065	6.26951036138809 7e-07	6.3632315034342 97e-07	0.001038	0.001124	0.000664 60943575 5904	0.000668787 9632950571
Linear Regression	0.000397	0.000453	0.00041175166829 9423	0.0004455504756 5838986	0.013064	0.013229	0.013733 43972079 3739	0.013883229 63331719
KNN	0.000007	0.000009	1.00216789861553 13e-05	1.0417744691715 05e-05	0.000498	0.000629	0.000599 35581809 40053	0.000698779 2619685491
SVR	0.001660	0.001689	4.56581118736526 e-05	4.3944370137617 304e-05	0.036915	0.037232	0.005926 09729821 2769 0.005903 53849902 086	0.005903538 49902086
NN-MLP	0.000063	0.000067	2.28165436487103 e-05	1.9307665210180 068e-05	0.005733	0.005723	0.003215 19284049 0915	0.003258958 4241428434

Figure 41: Table Showing Comparison between metric scores before and after Optimization

The comprehensive analysis presented in the above graphs and table underscores the substantial improvements achieved through hyperparameter tuning for each feature. This fine-tuning process has resulted in more optimal models, as evidenced by the significant reductions in both Mean Squared Error (MSE) and Mean Absolute Error (MAE).

Remarkably, these enhancements have led to distinct strengths in the performance of specific models. Following optimization, the Long Short-Term Memory (LSTM) model excels in minimizing MSE, signifying its proficiency in capturing nuanced patterns within the data. In contrast, the K-Nearest Neighbors (KNN) model demonstrates remarkable prowess in curbing MAE, underscoring its ability to make predictions with exceptional precision.

These outcomes collectively highlight the efficacy of hyperparameter tuning in enhancing the predictive capabilities of our models. It showcases the pivotal role that meticulous parameter adjustment plays in achieving superior results across different aspects of predictive accuracy.

8. Analysis of the AI project life cycle compliance with the AI ethics

Advancing AI responsibly demands proactive ethical considerations woven through all stages including problem framing, data management, model engineering, monitoring, and updating. A systemic approach is required across the project lifecycle **(Jobin et al., 2019).**

Problem Framing: The initial framing must critically assess whether AI technology application is appropriate and aligned with human values and welfare. Consulting diverse stakeholders including subject matter experts, legal and compliance teams, impacted communities, and cross-functional leadership uncovers potential pitfalls across security, privacy, fairness, accountability, and more. Their input also steers the effort towards equitable purposes respecting rights. Thoroughly documenting processes, risks discussed, and decisions made creates accountability trails for current and future teams.

Data Collection: Thoroughly vetting data sources, understanding origins and biases, obtaining informed consent where applicable, allowing data rights withdrawals, implementing privacy and confidentiality protections, employing techniques like data masking, and curating balanced samples that represent populations without disadvantaging groups demonstrates profound respect for people reflected in or impacted by the data. Comprehensive metadata detailing sources, consents, and handling allows for complete auditing. Focusing collection on datasets that actively foster social good and minimally risk harm guides principled sourcing.

Model Development: Incorporating technical explainability features like local interpretable model-agnostic explanations (LIME) sheds light on how models make decisions and build public trust through transparency. Rigorously testing for potential biases and harms using quantitative bias metrics and qualitative human-centered assessments prevents inadvertently discriminatory or unethical predictions. Fostering diversity, inclusion and multidisciplinary collaboration among the developers themselves flags more ethical issues early. Inclusive validation practices including adversarial testing against biases explicitly strengthen model performance across all groups **(Holstein et al., 2019).**

Deployment: Enabling human oversight in deployment for monitoring model use cases in the wild guards against unforeseen problems as societal contexts evolve. Automated update pipelines allow regular re-evaluations of inputs and outputs to maintain ethical performance even as populations and data patterns shift. User feedback channels including surveys, focus groups and participatory design facilitate continuous improvements based on diverse human experiences with the technology **(Raji et al., 2020).**

Cross-cutting Needs: Establishing multidisciplinary ethics boards, requiring explainability and fairness reports, performing impact assessments spanning data and algorithms, embracing comprehensive documentation, baking in emancipatory stakeholder input processes, and steadfastly prioritizing human flourishing provide ethical integrity across all AI project stages **(Floridi et al, 2018)**. Securing leadership buy-in, providing ethics education, and crafting customized guidelines anchored to core principles enact ethically aligned cultures.

In essence, responsible AI demands intentional, holistic thinking guided by fundamental shared values of justice, responsibility, transparency, privacy, inclusivity, and profound respect for human dignity. By embedding ethics intrinsically across the life cycle, AI can fully live up to its promise and possibilities to benefit humanity.

9. Conclusion

In summary, the website traffic prediction project has been a voyage encompassing data exploration, model development, and rigorous refinement. Employing a variety of machine learning algorithms, such as Linear Regression, K-Nearest Neighbors, Support Vector Regression, and Long Short-Term Memory networks, enabled the successful creation of predictive models.

While our achieved low Mean Squared Error and Mean Absolute Error can be attributed in part to the synthetic nature of our data and its limited features, it underscores the effectiveness of our models. Their remarkable performance stands as a testament to their adaptability and capability.

Moreover, the diligent pursuit of model optimization, achieved through meticulous hyperparameter tuning, substantially elevated predictive accuracy. Particularly, the Long Short-Term Memory (LSTM) model proved to be a stalwart, excelling in minimizing Mean Squared Error (MSE). Conversely, the K-Nearest Neighbors (KNN) model demonstrated remarkable precision in reducing Mean Absolute Error (MAE).

This holistic approach to website traffic prediction underscores the critical role of thoughtful model selection and fine-tuning in attaining precise forecasts. It further showcases the versatility of machine learning techniques in managing time series data for practical applications.

As we conclude this endeavour, it reaffirms the value of data-driven insights in guiding decision-making processes concerning website optimization and resource allocation. The journey, however, continues, holding the promise of further enhancements and revelations that can drive the success of online platforms and businesses in an ever-evolving digital landscape.

10. Recommendation

Real Data Collection: While this project utilized synthetic data, a valuable next step would be to implement the models with real-world website traffic data. This would provide more accurate insights and predictions, enabling businesses to make data-driven decisions.

Advanced Time Series Models: Investigate advanced time series forecasting models, such as Prophet or ARIMA. These models are specifically designed for handling temporal data and may capture complex patterns more effectively.

Scalability: Ensure that predictive models and infrastructure can scale efficiently to handle growing volumes of website traffic and data.

11. Future Work

As I reflect on the progress made in this project, I am enthusiastic about the opportunities for future work and improvements in the field of website traffic prediction. Here are some directions I would personally like to explore:

User Behavior Analysis: Incorporating user behavior data, such as click-through rates, session duration, and bounce rates, can lead to more comprehensive models for website traffic prediction.

Anomaly Detection: Developing algorithms for anomaly detection could be invaluable for identifying unusual traffic patterns, potentially indicating security threats or unexpected events.

Deployment as a Service: Transform the predictive models into a user-friendly service that businesses can integrate into their platforms for real-time traffic forecasting.

12. References:

A hybrid approach for web traffic prediction using deep learning algorithms. (2022, March 29). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/9772575

Anderson, A. et al. (2020). Synthetic Data for Model Robustness. Machine Learning Journal.

Brownlee, J. (2020). Hyperparameter optimization with random search and grid search. *MachineLearningMastery.com*. https://machinelearningmastery.com/hyperparameter-optimization-with-random-search-and-grid-search/

Choi, H., & Varian, H. R. (2012). Predicting the Present with Google Trends. *Economic Record*, *88*, 2–9. https://doi.org/10.1111/j.1475-4932.2012.00809.x

Crane, R. & Sornette, D. (2008). Robust dynamic classes revealed by measuring the response function Data Science. https://www.ijds.org/volume4/edition1/ijds2021.html

Davies, A. et al. (2020). Advances in Synthetic Time Series Generation. Machine Learning Journal.

How to create synthetic data sets for machine learning. (2021, March 4). https://practicaldatascience.co.uk/machine-learning/how-to-create-synthetic-data-sets-for-machine-learning

https://conferences.computer.org/annals/pdfs/annals_data_science_2022_paper_7.pdf

https://link.springer.com/article/10.1007/s10994-019-05855-6

https://link.springer.com/article/10.1007/s10994-020-05929-x

https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1475-4932.2012.00809.x

Jobin, A., & Ienca, M. (2019). The global landscape of AI ethics guidelines. *Nature Machine Intelligence*, *1*(9), 389–399. https://doi.org/10.1038/s42256-019-0088-2

Jones, F. & Lee, J. (2022). Data Augmentation Techniques for Time Series Data. Proceedings of the IEEE Data Science Conference.

of a social system. PNAS. https://www.pnas.org/content/105/41/15649

Perspectives, C. (2021). Web Traffic Time Series Predictions using LSTM & ARIMA Models. *www.clairvoyant.ai*. https://www.clairvoyant.ai/blog/web-traffic-time-series-predictions-using-lstm-and-arima-model#:~:text=obtained%20was%205.868.-,LSTM%20Model%3A,30%25%20as%20a%20test%20set.

Python, R. (2023). Linear regression in Python. *realpython.com*. https://realpython.com/linear-regression-in-python/

Sarker, I. H. (2021). Deep Learning: a comprehensive overview on techniques, taxonomy, applications and research directions. *SN Computer Science*, *2*(6). https://doi.org/10.1007/s42979-021-00815-1

Shaygan, M., Meese, C., Li, W., Zhao, X., & Nejad, M. (2022). Traffic prediction using artificial intelligence: Review of recent advances and emerging opportunities. *Transportation Research Part C-emerging Technologies*, *145*, 103921. https://doi.org/10.1016/j.trc.2022.103921

Smith, A. et al. (2021). Synthetic Data Generation for Time Series Forecasting. International Journal of

SVM Hyperparameter Tuning using GridSearchCV - Velocity Business Solutions Limited. (2020, February 19). Velocity Business Solutions Limited. https://www.vebuso.com/2020/03/svm-hyperparameter-tuning-using-gridsearchcv/

Tambe, V., Golait, A., Pardeshi, S., Javheri, R., & Arsalwad, G. (2022). Web Traffic Time Series Forecasting using ARIMA Model. *International Journal for Research in Applied Science and Engineering Technology*, *10*(5), 2447–2453. https://doi.org/10.22214/ijraset.2022.42800

Tan, P. et al. (2016). Introduction to Data Mining. Pearson.

Teixeira-Pinto, A. (2022, July 23). *2 K-Nearest Neighbours Regression | Machine Learning for biostatistics*. https://bookdown.org/tpinto_home/Regression-and-Classification/k-nearest-neighbours-regression.html

The grid search hyper-parameter sets for the CNN + LSTM model. (n.d.). ResearchGate. https://www.researchgate.net/figure/The-grid-search-hyper-parameter-sets-for-the-CNN-LSTM-model_tbl2_333939459

Yang, S. (2018, January 22). *Binary output layer of feedforward neural networks for solving multi-class classification problems*. arXiv.org. https://arxiv.org/abs/1801.07599