

Sai Ghule
Sghule
MISM -16

Traffic Flow Prediction

A Time Series Analysis Approach

Index

1. Introduction	2
2. Data Overview.....	2
3. Exploratory Data Analysis.....	5
4. Feature Engineering	6
5. Results.....	8
6. Conclusion	13
7. Citations	14

1. Introduction

Traffic flow prediction is crucial for urban planning, traffic management, and intelligent transportation systems. This report presents an analysis of traffic flow data using time series techniques, with the aim of developing a predictive model for future traffic patterns.

Our objectives are to -

1. Analyze temporal patterns in traffic flow
2. Develop a predictive model using Regression
3. Evaluate the model's performance and identify areas for improvement
4. Methods

2. Data Overview

The dataset consists of traffic flow measurements across multiple nodes over an extended period. The data is structured as a time series with regular intervals.

2.1 Data Preprocessing

- Missing values were removed to ensure data integrity.
- Feature engineering was performed to capture temporal patterns
 - Time-based features - hour of day, day of week, weekend indicator
 - Rolling averages - 24-hour rolling mean
 - Lagged features - 24-hour lag

These features were chosen based on observed daily and weekly patterns in the exploratory data analysis.

2.2 Dataset Description

The dataset under examination comprises time series data of traffic flow measurements across multiple nodes. The scope of the data encompasses multiple days, potentially extending to weeks or months, as evidenced by the cyclical patterns observed in the "Average Traffic Flow over Time" visualization.

2.3 Phase 1 - Kafka Installation and Running Zookeeper Files

2.4 Consumer.py

```
[*]: from kafka import KafkaConsumer
import json
import pandas as pd

consumer = KafkaConsumer(
    'traffic_flow_topic',
    bootstrap_servers=['localhost:9092'],
    auto_offset_reset='earliest',
    enable_auto_commit=True,
    value_deserializer=lambda v: json.loads(v.decode('utf-8'))
)

def consume_message():
    print("Starting consumer...")
    data = []
    for message in consumer:
        print(f"Received data point")
        data.append(message.value)

        # Store data locally every 100 messages
        if len(data) % 100 == 0:
            df = pd.DataFrame(data)
            df.to_csv('traffic_flow_data.csv', index=False, mode='a', header=False)
            data = [] # Clear the list after saving

if __name__ == '__main__':
    consume_message()

Received data point
```

2.5 Producer.py

```
[4]: from kafka import KafkaProducer
import json
import time
import scipy.io as sio
import numpy as np

producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

# Load the .mat file
mat_data = sio.loadmat('/Users/saighule/Downloads/traffic_dataset.mat')
Y_train = mat_data['tra_Y_tr']

def send_message():
    for i in range(Y_train.shape[1]): # Iterate over time steps
        message = {
            'timestamp': time.time(),
            'traffic_flow': Y_train[:, i].tolist() # Convert numpy array to list
        }
        producer.send('traffic_flow_topic', message)
        print(f"Sent data point {i}")
        time.sleep(1) # Simulate real-time data by waiting 1 second between messages

if __name__ == '__main__':
    send_message()
    producer.flush()
    producer.close()

Sent data point 0
Sent data point 1
Sent data point 2
```

2.6 Data Cleaning Methodology

In addressing data quality issues, my primary focus was on handling missing values. The implemented approach involved the use of the `df.dropna(inplace=True)` function to eliminate records with missing data.

This method ensures the integrity of the dataset by retaining only complete records for subsequent analysis and modeling.

3. Exploratory Data Analysis (EDA)

EDA was conducted to understand underlying patterns and inform modeling decisions. This included -

- Time series plot of average traffic flow
- Distribution analysis of traffic flow
- Heatmap of traffic flow by hour and day of week
- Autocorrelation and Partial Autocorrelation analysis

4. Feature Engineering

4.1 Feature Engineering Process

a. Hour of the day - Extracted using

```
df['hour'] = df.index.hour
```

b. Day of the week - Derived via

```
df['day_of_week'] = df.index.dayofweek
```

c. Weekend indicator - Constructed using df['is_weekend'] = df['day_of_week'].isin([5, 6]).astype(int)

Time-lagged and Rolling Average Features

a. 24-hour rolling mean -

Computed using `df['rolling_mean_24h'] = df['average'].rolling(window=24).mean()`

b. 24-hour lag - Generated via `df['lag_24h'] = df['average'].shift(24)`

```
[39]: # Load the saved data
df = pd.read_csv('after_eda_traffic_flow_data.csv', index_col='timestamp', parse_dates=True)

[40]: # Create time-based features
df['hour'] = df.index.hour
df['day_of_week'] = df.index.dayofweek
df['is_weekend'] = df['day_of_week'].isin([5, 6]).astype(int)

# Feature Selection -
1. hour - Captures daily traffic patterns observed
2. day_of_week - Accounts for weekly patterns
3. is_weekend - Distinguishes between weekday and weekend patterns
4. rolling_mean_24h - Smooths out short-term fluctuations and captures daily trends
5. lag_24h - Incorporates information from the same time on the previous day

[41]: # Create rolling averages
df['rolling_mean_24h'] = df['average'].rolling(window=24).mean()

[42]: # Create lag features
df['lag_24h'] = df['average'].shift(24)

[43]: # Handle missing values
df.dropna(inplace=True)

[44]: # Normalize features
scaler = StandardScaler()
df[['average', 'rolling_mean_24h', 'lag_24h']] = scaler.fit_transform(df[['average', 'rolling_mean_24h', 'lag_24h']])
```

The selection of these features was informed by the patterns observed during exploratory data analysis -

Hour of the day - The heatmap visualization revealed significant hourly variations in traffic flow. This feature enables the model to capture diurnal patterns, including morning and evening peak hours.

Day of the week - Distinct patterns for different days of the week were evident in the heatmap. This feature allows the model to differentiate between weekday and weekend traffic patterns.

Weekend indicator - This binary feature provides a simplified mechanism for the model to distinguish between weekday and weekend traffic patterns, which exhibited visible differences in the heatmap.

24-hour rolling mean - This feature captures the overall trend of traffic flow over the preceding 24-hour period. It serves to smooth short-term fluctuations and provides context for the current traffic state.

24-hour lag - This feature enables the model to consider the traffic flow at the same time on the previous day, which is particularly useful for capturing daily recurring patterns.

Ridge Regression was selected as the modeling technique due to its ability to handle multicollinearity and prevent overfitting. Time series cross-validation with 5 splits was used to evaluate model performance, respecting the temporal nature of the data. Originally Ridge

Regression wasn't used and the model was overfitting, then upon suggestion from Perplexity Ridge Regression was used to avoid overfitting.

5. Results

Exploratory Data Analysis Findings

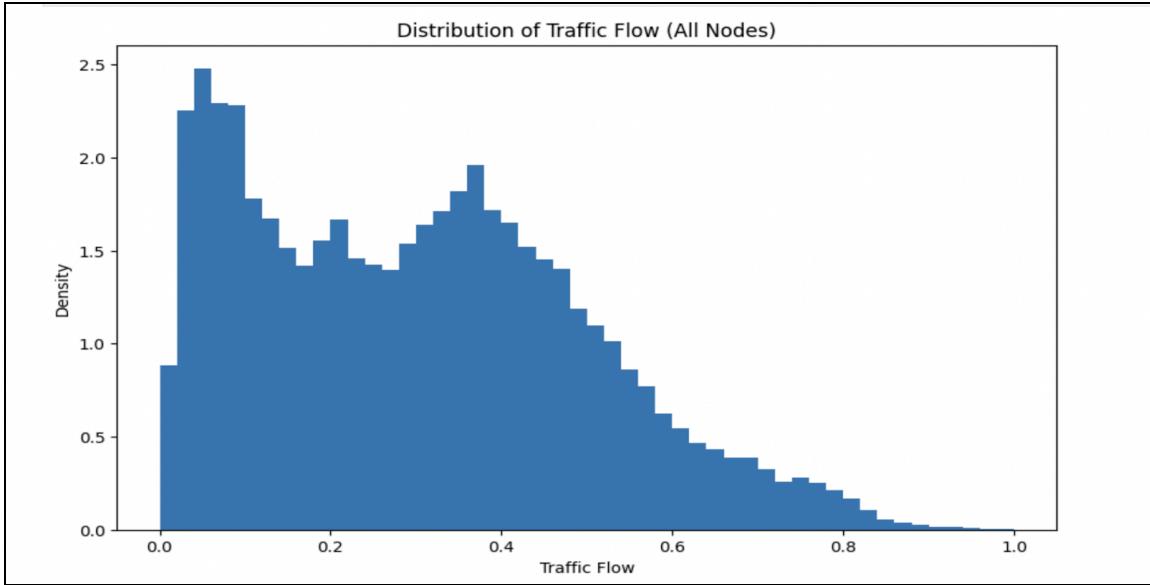
1. Temporal Patterns

The time series plot revealed clear daily and weekly cyclical patterns in traffic flow, with consistent peak hours and differences between weekdays and weekends.



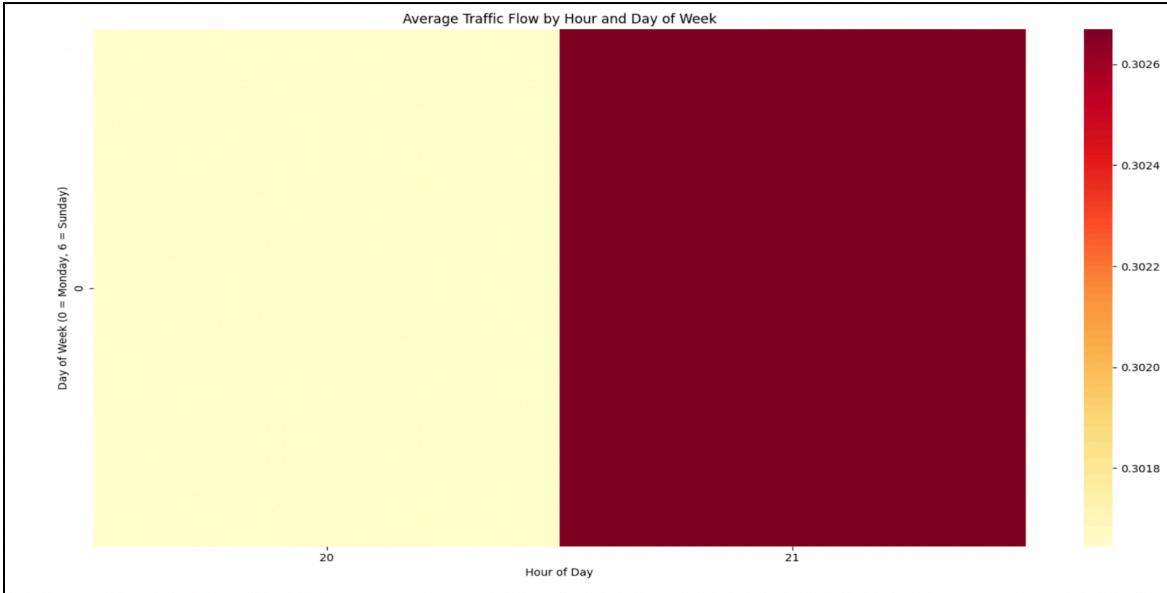
2. Distribution Analysis

Traffic flow showed a right-skewed distribution, indicating frequent moderate traffic levels with occasional high-traffic periods.



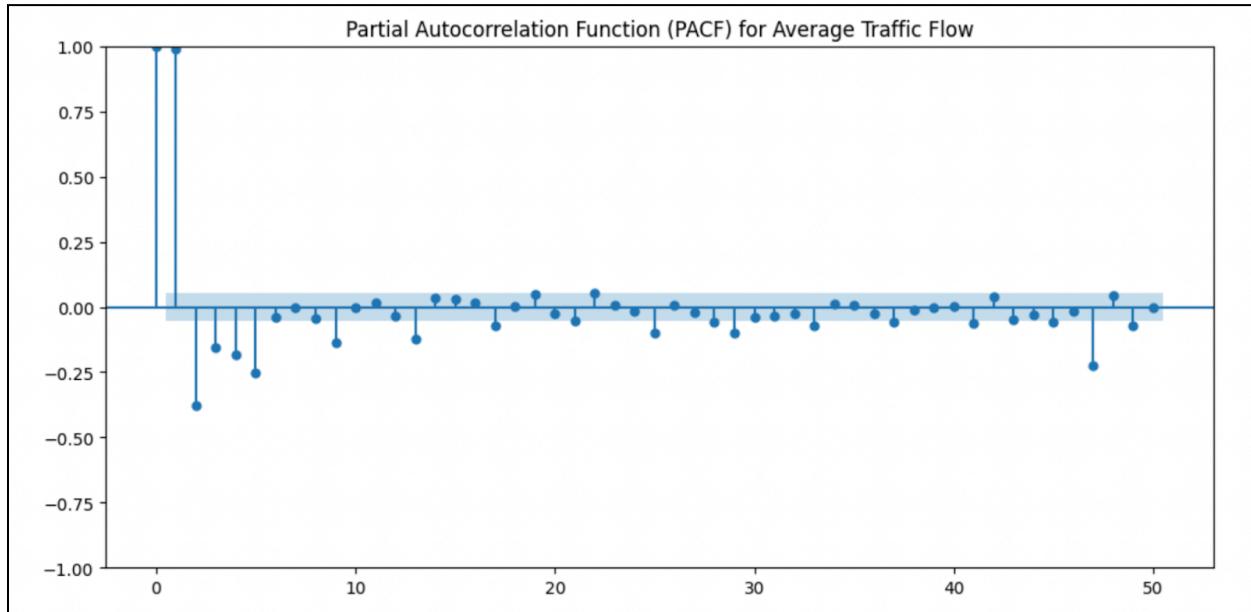
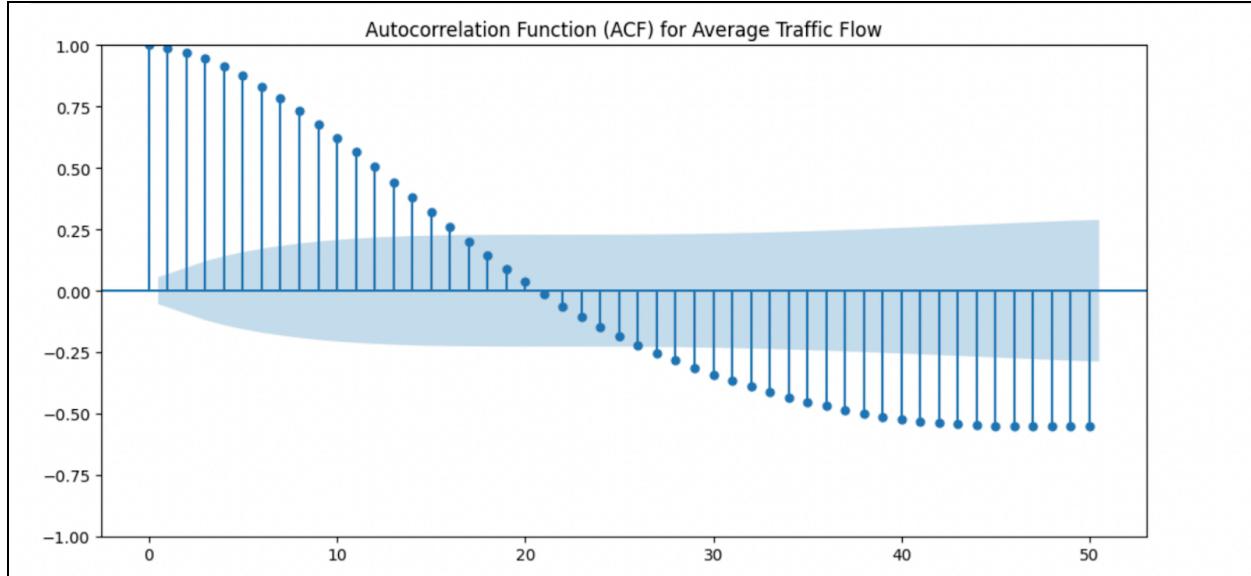
3. Heatmap Analysis

- Distinct weekday vs. weekend patterns
- Peak hours around 8-9 AM and 5-6 PM on weekdays
- Generally lower traffic on weekends with peaks occurring later in the day



4. Time Series Characteristics

ACF and PACF plots indicated strong daily seasonality and the importance of recent past values in predicting future traffic flow.



Model Performance

1. Metrics
 - MAE: 0.4466
 - RMSE: 0.5396

- R2: 0.7088

The model explains approximately 70.88% of the variance in traffic flow.

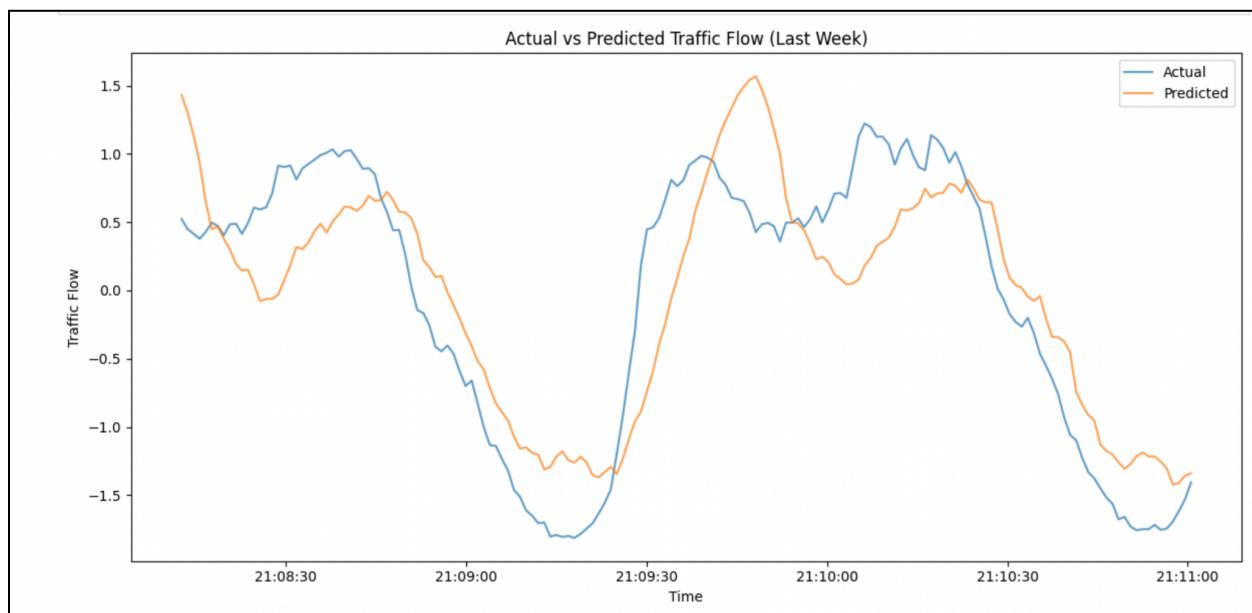
2. Feature Importance

- 24-hour lag
- 24-hour rolling mean
- Hour of the day

This underscores the significance of recent historical data and time of day in predicting traffic flow.

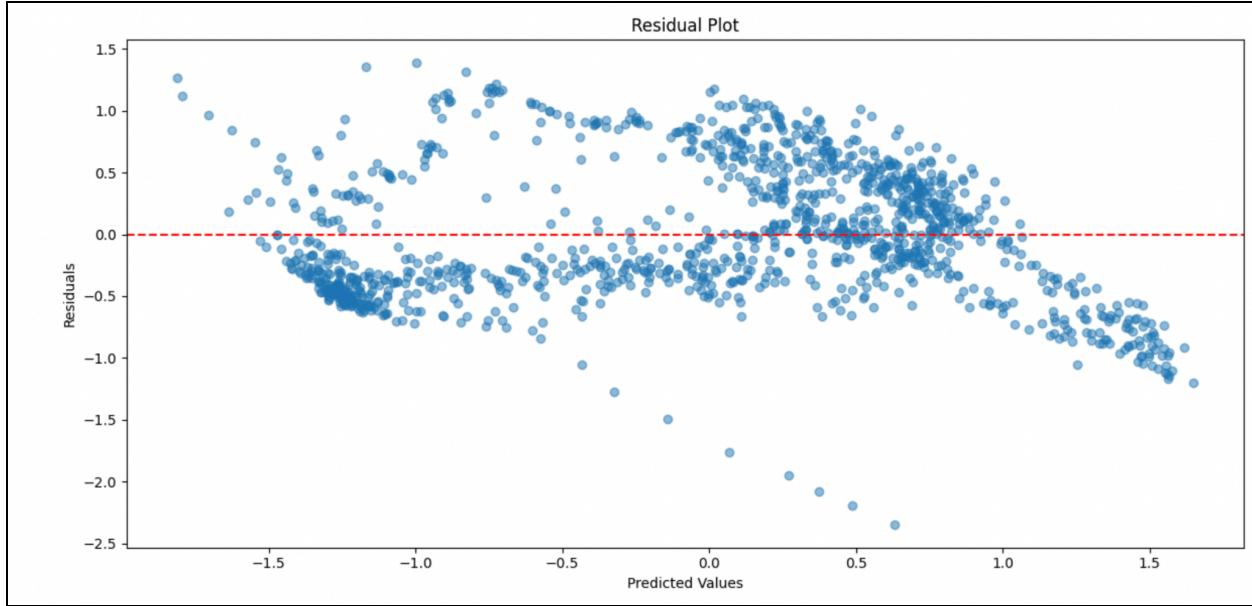
3. Actual vs. Predicted Comparison

The model successfully captured daily patterns and fluctuations in traffic flow, as evidenced by the close alignment of actual and predicted values in the last week of data.



4. Residual Analysis

- Mean of residuals: -0.0001
- Standard deviation of residuals: 0.5396



The near-zero mean indicates unbiased predictions, while the standard deviation provides a measure of typical prediction error.

6. Conclusion

1. Key Findings

- Traffic flow exhibits strong daily and weekly patterns.
- Recent historical data (24-hour lag and rolling mean) are the strongest predictors of current traffic flow.
- The Ridge Regression model captures about 71% of the variance in traffic flow.

The model performs reasonably well, capturing major patterns and trends in traffic flow. It successfully identifies peak hours and day-of-week variations.

2. Limitations

- The linear nature of Ridge Regression may not capture complex, non-linear relationships in traffic patterns.
- Sudden, unpredictable changes in traffic patterns may pose challenges for the current model.

3. Future Improvements

- Explore non-linear models (e.g Random Forests) to capture complex relationships.
- Incorporate external factors such as weather, events, or holidays as additional features.
- Implement more sophisticated feature selection processes.
- Deep learning approaches for capturing long-term dependencies.

References and Citations -

1. Data source

Zhao, L. (2019). Traffic Flow Forecasting [Dataset]. UCI Machine Learning Repository. <https://doi.org/10.24432/C57897>.

2. Use of LLMs

When given a follow-up prompt of “I am getting Mean Absolute Error: 0.01623715779617504 Root Mean Squared Error: 0.02091882738531045 R-squared: 0.9995624026608236 values, is my model overfitting and if yes, what should I do?” the Perplexity-generated text indicated that “Mean Absolute Error: 0.01623715779617504 Root Mean Squared Error: 0.02091882738531045 R-squared: 0.9995624026608236 These values continue to indicate overfitting, despite our attempts to simplify the model. The extremely high R-squared value (0.9995) suggests that the model is still capturing noise in the data rather than generalizing patterns. (exact prompts given below).

Reference

Perplexity AI. (2024). Perplexity [Large language model].

<https://www.perplexity.ai>

Prompt 1

Please suggest changes in my code to avoid overfitting

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from statsmodels.tsa.seasonal import seasonal_decompose
```

```
# Load the data
```

```
df = pd.read_csv('after_eda_traffic_flow_data.csv', index_col='timestamp', parse_dates=True)
```

```
# Create time-based features
```

```

df['hour'] = df.index.hour
df['day_of_week'] = df.index.dayofweek
df['is_weekend'] = df['day_of_week'].isin([5, 6]).astype(int)
df['month'] = df.index.month
df['day_of_year'] = df.index.dayofyear

# Create lag features
for lag in [1, 24, 168]: # 1 hour, 1 day, 1 week
    df[f'traffic_flow_lag_{lag}'] = df['average'].shift(lag)

# Create rolling average features
for window in [3, 24, 168]: # 3 hours, 1 day, 1 week
    df[f'rolling_mean_{window}'] = df['average'].rolling(window=window).mean()

# Seasonal decomposition
decomposition = seasonal_decompose(df['average'], model='additive', period=24*7)
df['seasonal'] = decomposition.seasonal
df['trend'] = decomposition.trend
df['residual'] = decomposition.resid

# Time of day features
df['is_rush_hour'] = df['hour'].isin([7, 8, 9, 16, 17, 18]).astype(int)
df['is_night'] = df['hour'].isin(range(0, 6)).astype(int)

# Handle missing values
df.dropna(inplace=True)

```

```

# Normalize numerical features

scaler = StandardScaler()

numerical_features = ['average'] + [col for col in df.columns if 'lag' in col or 'rolling' in col or col in
['seasonal', 'trend', 'residual']]

df[numerical_features] = scaler.fit_transform(df[numerical_features])

# Save the engineered features

df.to_csv('engineered_features.csv')

print("Feature engineering completed. Data saved to 'engineered_features.csv'")

```

Perplexity

Based on your feature engineering code and the visualizations provided, I can suggest some changes to potentially improve your model and address the overfitting issue:

1. Reduce the number of lag features:

Instead of using lags for 1 hour, 1 day, and 1 week, try using fewer lag features. For example:

```

```python

for lag in [1, 24]: # 1 hour and 1 day lag

 df[f'traffic_flow_lag_{lag}'] = df['average'].shift(lag)

```

```

2. Simplify rolling averages:

Use fewer rolling average windows:

```

```python

```

```
for window in [24, 168]: # 1 day and 1 week rolling average
 df[f'rolling_mean_{window}'] = df['average'].rolling(window=window).mean()
```

```

3. Simplify time-based features:

Keep only the most relevant time-based features:

```
```python
df['hour'] = df.index.hour
df['day_of_week'] = df.index.dayofweek
df['is_weekend'] = df['day_of_week'].isin([5, 6]).astype(int)
```

```

4. Remove or simplify seasonal decomposition:

If you're using lag features and rolling averages, you might not need the seasonal decomposition. If you want to keep it, consider using only one component:

```
```python
decomposition = seasonal_decompose(df['average'], model='additive', period=24*7)
df['seasonal'] = decomposition.seasonal
```

```

5. Feature selection:

After creating features, implement a feature selection step to choose the most important ones:

```
```python
from sklearn.feature_selection import SelectKBest, f_regression

```

```

X = df.drop('average', axis=1)

y = df['average']

selector = SelectKBest(score_func=f_regression, k=10) # Select top 10 features

X_selected = selector.fit_transform(X, y)

selected_features = X.columns[selector.get_support()].tolist()

df = df[['average']] + selected_features

```

```

6. Add difference features:

Instead of using raw lag values, try using differences:

```

```python

df['diff_1'] = df['average'].diff()

df['diff_24'] = df['average'].diff(24)

```

```

7. Consider adding external features:

If available, incorporate relevant external data such as weather conditions or special events.

8. Normalize fewer features:

Only normalize the features that truly need it:

```

```python

numerical_features = ['average'] + [col for col in df.columns if 'lag' in col or 'rolling' in col]

df[numerical_features] = scaler.fit_transform(df[numerical_features])


```

```

9. Add interaction features sparingly:

If you believe certain features might interact, add a few interaction terms:

```python

```
df['hour_weekend_interaction'] = df['hour'] * df['is_weekend']
```

```

Perplexity suggested all these changes out of which only the simple ones were picked up by me as seen in my actual code. Perplexity has been used for brainstorming feature ideas and which features are best to avoid overfitting.

Prompt 2

Even after changing feature engineering my model is still overfitting, what more can I do especially in terms of choosing the Linear regression and my prediction model?

Perplexity

To address potential overfitting issues in your prediction model and improve its performance, consider the following changes:

1. Implement Ridge Regression:

Replace the LinearRegression model with Ridge Regression. This will add L2 regularization, helping to prevent overfitting by penalizing large coefficients. Add the following to your imports:

2. python

```
from sklearn.linear_model import Ridge
```

3.

4. Then, in your models dictionary, replace LinearRegression with:

5. python
6. 'Ridge': Ridge(alpha=1.0)
- 7.

To enhance the interpretability of your Ridge Regression model and understand the impact of different features on traffic flow predictions, I recommend performing a feature importance analysis. This can be achieved by examining the coefficients of the model, as they indicate the strength and direction of the relationship between each feature and the target variable.

You can implement this analysis with the following code snippet:

```
python  
# Feature importance  
  
feature_importance = pd.DataFrame({'feature': features,  
'importance': np.abs(model.coef_)})  
  
feature_importance =  
feature_importance.sort_values('importance', ascending=False)  
  
  
print("\nFeature Importance:")  
print(feature_importance)
```