LSTM – ARIMA Code Explanation

Preprocessing File

import libraries

```
import warnings
warnings.filterwarnings('ignore')
```

```
import pandas as pd
from tqdm import tqdm
import tensorflow as tf
import matplotlib.pyplot as plt
```

Explanation: The above code snippets are based on two cells. The first cell imports the warning library and ignores all the warnings for display in jupyter notebook. The second cell import the required libraries including the pandas, TQDM, TensorFlow and matplotlib for the proposed work.

Read Dataset

```
df = pd.read_csv('data/combined.csv', header=None)
print(df.shape)
df.head(2)
```

```
(186559, 20)
```

	0	1	2	3	4	5	6	7	8	
0	2940388;2022-02-21;2022-02-21;16:44:34;16:46:00;3	010;2100	000;1	433;142	3;140	2;A;59;A59;oplopend;Rosmalen;Rosmalen-Oost;'s-...	NaN	NaN	NaN	
1	2940390;2022-02-21;2022-02-21;16:46:33;18:01:4...	627;2895	000;75	183;56	3;58	6;A;16;A16;aflopend;Breda-Noord;Markbrug;Breda...	[Defecte vrachtwagen(s) 67]	[Geen oorzaakcode opgegeven door VWM 3];[000]	[HBD]	[vracht

Explanation: The above code read the dataset using the read_csv function of the pandas library. Further, it printed the number of rows and columns of the dataset by using the shape function. Lastly, the first two samples of the dataset were printed by using the head function.

```
columns_str = 'NLSitNummer;DatumFileBegin;DatumFileEind;TijdFileBegin;TijdFileEind;FileZwaarte;GemLengte;FileDuur;HectometerKop;H
columns = columns_str.split(';')
print(len(columns))
columns
```

Explanation: We copy the columns name from the head of the first column and save it in the columns_str variable. Further, the string represents that it is based on the name of all columns in the dataset. The string was split by ';' that returns the column names in the form of a list by splitting the string. In the third line, the length of the extracted columns name was calculated by using the len() function. In the last line, the column names were printed in the file.

```
df[['NLSitNummer',
 'DatumFileBegin',
 'DatumFileEind',
 'TijdFileBegin',
 'TijdFileEind',
 'FileZwaarte']] = df[0].apply(lambda x: pd.Series(str(x).split(";")[:6]))

df[['FileDuur']] = df[2].apply(lambda x: pd.Series(str(x).split(";")[-1]))

df[['unknown',
    'RouteLet',
 'RouteNum',
 'RouteOms',
 'hectometreringsrichting',
 'KopWegvakVan',
 'KopWegvakNaar',
 'TrajVan',
 'TrajNaar']] = df[5].apply(lambda x: pd.Series(str(x).split(";")[:9]))
```

Explanation: The columns in the original dataset were based on multiple values. The dataset was processed to split the values into separate columns. For this purpose, it was seen that the first column has six values for each sample. In the first line, the first column was split by ';' into six different columns and then store in separate columns as shown in line 1.

The second column was not necessary that was skipped. The third column consisted of the two values and the second value was useful. The second in the above code extracted the values of third column and store the useful 'FileDuur' value in separate column.

In the last line, the sixth column that also have the multiple values in each cell was split by ';' and store the extracted 9 values into separate columns.

```
df = df[['NLSitNummer',
'DatumFileBegin',
'DatumFileEind',
'TijdFileBegin',
'TijdFileEind',
'FileZwaarte',
'FileDuur',
'RouteLet',
'RouteNum',
'RouteOms',
'hectometreringsrichting',
'KopWegvakVan',
'KopWegvakNaar',
'TrajVan',
'TrajNaar']]

print(df.shape)
df.head()
```

```
(186559, 15)
```

	NLSitNummer	DatumFileBegin	DatumFileEind	TijdFileBegin	TijdFileEind	FileZwaarte	FileDuur	RouteLet	RouteNum	RouteOms	hectometreringsrichting
0	2940388	2022-02-21	2022-02-21	16:44:34	16:46:00	3	1	A	59	A59	oplopend
1	2940390	2022-02-21	2022-02-21	16:46:33	18:01:44	217	75	A	16	A16	aflopend

Explanation: In the above code snippet, the extracted columns after preprocessing that have a single value were fetched from the data frame. Further, the shape of the data frame was calculated. Lastly, the first five samples of the data frame were printed using the head() function.

```
states = ['Breda', 'Tilburg']
df = df[df.TrajVan.isin(states)]
df = df[df.TrajNaar.isin(states)]

print(df.shape)
```

```
(5365, 15)
```

Explanation: According to the requirements, the processed data-frame was filtered for the samples that the starting state or ending state was Breda or Tilburg. In the above code the, second and third line filter the samples of Breda and Tilburg based on the starting and ending states.

```
df_B_T = df[df.TrajVan == states[0]]
df_T_B = df[df.TrajVan == states[1]]

print(df_B_T.shape)
print(df_T_B.shape)
```

```
(1985, 15)
(3380, 15)
```

Explanation: Further, the filtered dataset was split into two different data frames. In the first line the samples that the starting state from Breda was stored in a separate data-frame labelled as df_B_T. In the second line, the samples that have the starting state is Tilburg were stored in df_T_B. Lastly, the shape of both data-frame was printed using the shape function.

```
df_B_T[["DatumFileBegin", "DatumFileEind", "TijdFileBegin", "TijdFileEind"]] = df_B_T[["DatumFileBegin", "DatumFileEind", "TijdFi
df_T_B[["DatumFileBegin", "DatumFileEind", "TijdFileBegin", "TijdFileEind"]] = df_T_B[["DatumFileBegin", "DatumFileEind", "TijdFi

df_B_T = df_B_T.sort_values(by=['DatumFileBegin', 'TijdFileBegin'])
df_T_B = df_T_B.sort_values(by=['DatumFileBegin', 'TijdFileBegin'])
```

Explanation: As the dataset was not properly distributed according to the time. The type of Starting Date, Ending Date, start time and end time columns were converted into DateTime type. Further, the datasets were sorted based on starting date and time in last two lines of the above code snippet.

```
df_plot = df_B_T[['FileDuur']]
df_plot = df_plot.set_index(df_B_T.DatumFileBegin)
df_plot['FileDuur'] = df_plot['FileDuur'].astype(str).astype(int)
df_plot.plot()
```

```
df_plot = df_T_B[['FileDuur']]
df_plot = df_plot.set_index(df_T_B.DatumFileBegin)
df_plot['FileDuur'] = df_plot['FileDuur'].astype(str).astype(int)
df_plot.plot()
```

Explanation: The above two code snippets were used to plot the data on the monthly basis. In these snippets, the traffic jam column was fetched and set as indexed first. Further, it was plotted using the plot function of the matplotlib library.

```
df_B_T['starting_hour'] = df_B_T['TijdFileBegin'].dt.hour
df_B_T['starting_minute'] = df_B_T['TijdFileBegin'].dt.minute
df_B_T['Ending_hour'] = df_B_T['TijdFileEind'].dt.hour
df_B_T['Ending_minute'] = df_B_T['TijdFileEind'].dt.minute

df_T_B['starting_hour'] = df_T_B['TijdFileBegin'].dt.hour
df_T_B['starting_minute'] = df_T_B['TijdFileBegin'].dt.minute
df_T_B['Ending_hour'] = df_T_B['TijdFileEind'].dt.hour
df_T_B['Ending_minute'] = df_T_B['TijdFileEind'].dt.minute
```

Explanation: To convert the dataset into records on an hourly basis, it is necessary to separately get the date, hour and minutes. For this purpose, the starting time and ending time column of both dataset was split into hour and minute column using the above code snippet.

```
def dataset_hourly_mapping(raw_df, state_from, state_to, start_date='2022-02-01', end_date='2022-10-01'):
    pending_score = 0
    data_rows = []
    jem_times = []

    dates = pd.Series(pd.date_range('2022-02-01', '2022-10-01', freq='h'))
    df_dates = pd.DataFrame({'Date':dates})

    for dd in tqdm(list(df_dates['Date'])):
        filtered_df = raw_df[(raw_df['DatumFileBegin'].dt.date==dd.date()) & (raw_df['starting_hour']==dd.hour)]

        if filtered_df.shape[0] > 0:
            jem_time = 0 if pending_score == 0 else pending_score
            for index, row_item in filtered_df.iterrows():
                if row_item.starting_hour == row_item.Ending_hour:
                    jem_time += row_item.Ending_minute - row_item.starting_minute
                else:
                    jem_time += 60 - row_item.starting_minute
                    pending_score = row_item.Ending_minute
            data_rows.append([dd, jem_time, state_from, state_to])
        else:
            if pending_score != 0:
                jem_time = pending_score
                data_rows.append([dd, jem_time, state_from, state_to])
                pending_score = 0
            else:
                data_rows.append([dd, 0, state_from, state_to])
    return data_rows
```

Explanation: The above code snippet is based on the customize function that receives the data-frame and states with date range. Further, it creates a new data-frame that has the DateTime column with a variance of one hour. Further, the customize function iterate over the generated data frame (df_dates). It matches the date and time from our processed dataset and calculates the total time of traffic jams in the current specific hour at a specific date. After calculating the traffic jam in every hour between the provided date range, it returns the calculated data-frame.

```
data_rows = dataset_hourly_mapping(df_B_T, 'Breda', 'Tilburg')
df_hourly_B_T = pd.DataFrame(data=data_rows, columns=['Date', 'Jem_Time', 'State_from', 'State_to'])
df_hourly_B_T.to_csv('df_hourly_B_T.csv', index=False)

data_rows = dataset_hourly_mapping(df_T_B, 'Tilburg', 'Breda')
df_hourly_T_B = pd.DataFrame(data=data_rows, columns=['Date', 'Jem_Time', 'State_from', 'State_to'])
df_hourly_T_B.to_csv('df_hourly_T_B.csv', index=False)
```

```
100%|| 5809/5809 [00:07<00:00, 760.79it/s]
100%|| 5809/5809 [00:10<00:00, 558.95it/s]
```

Explanation: The above code snippet uses the customize function for both datasets. After calling the customize function, the first line gets back the processed samples. Further, the processed samples were converted in data-frame in the second line. In the third line, the finalized dataset was saved using the to_csv function of pandas. The same steps were followed for the second dataset. After preparing both datasets, the datasets were merged into a single data frame as shown in the below code snippet.

```
finalized_df = pd.concat([df_hourly_B_T, df_hourly_T_B], ignore_index=True)
```

```
finalized_df = finalized_df.sort_values(by=['Date', 'State_from'])
```

LSTM_ARIMA File

```
import pandas as pd
from tqdm import tqdm
import tensorflow as tf
import matplotlib.pyplot as plt
```

```
# Start by importing the necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.arima_model import ARIMA
```

```
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from math import sqrt
from matplotlib import pyplot
import numpy
```

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
# from keras.models import Sequential
# from keras.layers import LSTM, Dense
```

Explanation: The above cells import the required libraries for the manipulation of the dataset, training and testing of the LSTM and ARIMA model.

```
finalized_df = pd.read_csv('/finalized.csv')
```

```
finalized_df = finalized_df[['Date', 'State_from', 'State_to', 'Jem_Time']]
finalized_df.head()
```

	Date	State_from	State_to	Jem_Time
0	2022-02-01 00:00:00	Breda	Tilburg	0
1	2022-02-01 00:00:00	Tilburg	Breda	0
2	2022-02-01 01:00:00	Breda	Tilburg	0
3	2022-02-01 01:00:00	Tilburg	Breda	0
4	2022-02-01 02:00:00	Breda	Tilburg	0

The first cell read the processed dataset that was saved in the previous file. In the second cell, the date, states, and Jem time column were fetched and viewed by the head function.

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
finalized_df['State_from'] = le.fit_transform(finalized_df['State_from'])
finalized_df['State_to'] = le.transform(finalized_df['State_to'])
```

As the states column has the two states Breda and Tilburg after the processing of the dataset. For the training of the model, all the columns should be integral values rather than categorical values. For this purpose, the Label Encoder function of scikit-learn library was used. It converts categorical values into integral values.

In the above code snippet, firstly the Label Encoder was initialized. In the last two lines, it was used to fit the state-from and state-to columns that convert the categorical value into a numeric representation.

```
# Make sure the data is a time series
finalized_df.index = pd.to_datetime(finalized_df['Date'])
finalized_df = finalized_df[['State_from', 'State_to', 'Jem_Time']]
finalized_df.head()
```

Date	State_from	State_to	Jem_Time
2022-02-01 00:00:00	0	1	0
2022-02-01 00:00:00	1	0	0
2022-02-01 01:00:00	0	1	0
2022-02-01 01:00:00	1	0	0
2022-02-01 02:00:00	0	1	0

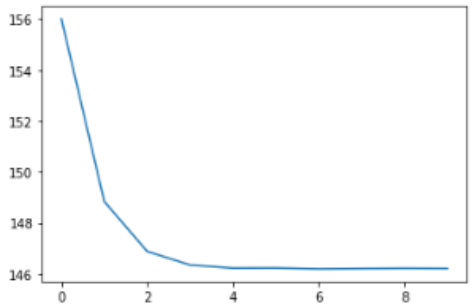
The date column was set as the index in the first line of the above code. The rest of the columns were fetched once again and displayed by the head function. Now you can notice that the state-from and state-to columns have been converted into the numerical representation.

```
# Extract input and output variables
X = finalized_df.iloc[:, 1:-1]
y = finalized_df.iloc[:, -1]

# Normalize the data using MinMaxScaler
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
# y = scaler.fit_transform(y)

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0, shuffle=False)

# Reshape the data for LSTM input
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))

# Define the LSTM model
model = Sequential()
model.add(LSTM(50, activation='tanh', input_shape=(X_train.shape[1], 1)))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')

# Train the model
history = model.fit(X_train, y_train, epochs=10, batch_size=32, verbose=1)

# Evaluate the model on the test set
score = model.evaluate(X_test, y_test, verbose=0)

# Make predictions on the test set
y_pred = model.predict(X_test)
```

The above code snippet is already commented in the code file, but I will explain it line by line here. Firstly, the features were extracted in the X variable and the target column in y variable. Further, the dataset was normalized using the min-max-scaler function that normalized the dataset. In the third step, the dataset was split into train and test sets with a ratio of 70% and 30% respectively using the train-test-split function of scikit-learn library. Next, the train and test features were reshaped to fulfil the requirements of the LSTM model. Further, the LSTM model was developed by adding the LSTM layer and output layer. The model was compiled by setting the loss function as MSE.

In the third last step, the model was fit for training using the training features and train labels. Further, the test features and labels were used to calculate the score of the trained LSTM model. In the last step, the test features were used to make the predictions with the trained model.

```
# plot metrics
pyplot.plot(history.history['loss'])
pyplot.show()
```




```
# Calculate the root mean squared error
rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print("Root mean squared error: {:.2f}".format(rmse))
```

The first cell plots the graph of loss during the training of the model. The second cell calculate the loss by using the predicted traffic jam and actual traffic jam. The last line prints the RMSE of trained LSTM model using the test samples.

```
import pmdarima as pm
```

```
data = finalized_df
```

```
data = data[['State_from', 'State_to', 'Jem_Time']]
data.head()
```

Date	State_from	State_to	Jem_Time
2022-02-01 00:00:00	0	1	0
2022-02-01 00:00:00	1	0	0
2022-02-01 01:00:00	0	1	0
2022-02-01 01:00:00	1	0	0
2022-02-01 02:00:00	0	1	0

The code of the ARIMA model starts from here as already commented in the code. The first line imports the library. The second line saved the finalized dataset into the data variable. The second last line fetches the features and traffic jam column and displays it using the head function.

```
data = data[data['State_from'] == 0]
data.head()
```

Date	State_from	State_to	Jem_Time
2022-02-01 00:00:00	0	1	0
2022-02-01 01:00:00	0	1	0
2022-02-01 02:00:00	0	1	0
2022-02-01 03:00:00	0	1	0
2022-02-01 04:00:00	0	1	0

Filtered the samples that the root from Breda to Tilburg only.

```
train_size = int(len(data) * 0.8)
test_size = len(data) - train_size
```

```
# Split the data into train and test sets
train, test = data[0:train_size], data[train_size:]
train.head()
```

Once again split the dataset into training and testing sets with the ratio of 80% and 20% respectively.

```
model = pm.auto_arima(data['Jem_Time'],
                      m=12, seasonal=True,
                    start_p=0, start_q=0, max_order=4, test='adf',error_action='ignore',
                        suppress_warnings=True,
                    stepwise=True, trace=True)
```

```
Performing stepwise search to minimize aic
 ARIMA(0,0,0)(1,0,1)[12] intercept   : AIC=inf, Time=10.37 sec
 ARIMA(0,0,0)(0,0,0)[12] intercept   : AIC=41285.799, Time=0.12 sec
 ARIMA(1,0,0)(1,0,0)[12] intercept   : AIC=39798.358, Time=4.64 sec
 ARIMA(0,0,1)(0,0,1)[12] intercept   : AIC=40057.091, Time=5.03 sec
 ARIMA(0,0,0)(0,0,0)[12]             : AIC=41753.688, Time=0.07 sec
 ARIMA(1,0,0)(0,0,0)[12] intercept   : AIC=39802.935, Time=0.30 sec
 ARIMA(1,0,0)(2,0,0)[12] intercept   : AIC=39676.159, Time=15.43 sec
 ARIMA(1,0,0)(2,0,1)[12] intercept   : AIC=inf, Time=44.15 sec
 ARIMA(1,0,0)(1,0,1)[12] intercept   : AIC=inf, Time=16.76 sec
 ARIMA(0,0,0)(2,0,0)[12] intercept   : AIC=40995.488, Time=11.57 sec
 ARIMA(2,0,0)(2,0,0)[12] intercept   : AIC=39677.799, Time=21.00 sec
 ARIMA(1,0,1)(2,0,0)[12] intercept   : AIC=39677.829, Time=19.87 sec
 ARIMA(0,0,1)(2,0,0)[12] intercept   : AIC=39897.376, Time=13.70 sec
 ARIMA(2,0,1)(2,0,0)[12] intercept   : AIC=39680.083, Time=25.72 sec
 ARIMA(1,0,0)(2,0,0)[12]             : AIC=39805.965, Time=5.03 sec

Best model:  ARIMA(1,0,0)(2,0,0)[12] intercept
Total fit time: 194.045 seconds
```

Initialized the ARIMA model and passed the traffic jam column for the best fit. You can notice here; the LSTM model is trained on the multi-variant technique in which we passed the features with the target variable while the ARIMA model is trained on the uni-variant technique in which we only passed the target variable.

```
model.fit(train['Jem_Time'])
```

```
ARIMA(order=(1, 0, 0), scoring_args={}, seasonal_order=(2, 0, 0, 12),
      suppress_warnings=True)
```

```
forecast=model.predict(n_periods=test_size, return_conf_int=True)
```

After finding the best fit for the data, the model was fit for the training in the first cell. In the second cell the predictions were made with the trained ARIMA model equal to the test size

```
forecast_df = pd.DataFrame(forecast[0],index = test.index,columns=['Prediction'])
```

```
forecast_df
```

	Prediction

Prepare a data-frame with the predictions of the ARIMA model in the first cell. In the second cell, just display the data frame.

```
pd.concat([data['Jem_Time'],forecast_df],axis=1).plot()
```

The above code cell merges the actual data with predicted data and plots it to see the trend.

```
predictions =  list(predictions)
y_test = list(test['Jem_Time'])

# Calculate the root mean squared error
rmse = np.sqrt(mean_squared_error(y_test, predictions))
print("Root mean squared error: {:.2f}".format(rmse))
```

The above code snippet uses the predictions and after converting them into a list, stores them in the 'predictions' variable. Similarly, convert the traffic jam column into a list.

Further, the above code used both lists to calculate the RMSE and lastly print it.

Streamlit – Functions File

```
def get_df(state_from_index, state_to_index):
    today_date = date.today()
    end_date = today_date + timedelta(15)
    today_date, end_date = str(today_date), str(end_date)
    forecast_range = pd.date_range(start=today_date, end=end_date, freq='H')

    state_from = [state_from_index] * len(forecast_range)
    state_to = [state_to_index] * len(forecast_range)

    unseen_df = {
        'Date': forecast_range,
        'State_from': state_from,
        'State_to': state_to
    }

    df1 = pd.DataFrame(unseen_df)
    df1 = df1.set_index('Date')
    return df1
```

This function receives the starting state and ending state as the parameter value. In the first line, it gets today's data. Further, it adds the 15 days to today's date and stores it in the end-date variable. Further, prepare a list that contains the values starting with today's date and ending with the date 15 days ahead with hourly distribution. Next, Prepare the columns equal to the length of the date-range list for starting and ending state values. Append the date, stating and ending state list in one data frame label as unseen-df. After that, the date column was set as the index column of the data frame.

```
def main(state_from_index, state_to_index):
    scaler_filename = "models/scaler.pkl"
    scaler = joblib.load(scaler_filename)
    lstm_model = load_model('models/my_model.h5')

    df1 = get_df(state_from_index, state_to_index)
    transform_df = scaler.transform(df1)
    reshaped_df = np.reshape(transform_df, (transform_df.shape[0], transform_df.shape[1], 1))
    predictions = lstm_model.predict(reshaped_df)
    predictions = normalize_predictions(predictions)

    df1['Traffic Jam'] = predictions

    df1.reset_index(inplace=True)
    df1 = df1.rename(columns={'index': 'Date'})

    df1 = df1[df1.Date.dt.hour >= 7]
    df1 = df1[df1.Date.dt.hour < 19]

    working_days = [0, 1, 2, 3, 4]
    df1['day'] = df1['Date'].dt.dayofweek
    df1 = df1[df1['day'].isin(working_days)]
    print(df1)
```

This function also receives the starting and ending state as the parameter value. Further, it loads the normalizer label as min-max-scaler and the trained LSTM model. In the 4th line, it passes the index of starting state and ending state to the previous function and gets the data frame. The above function scaled and reshaped the fetched data frame to fulfil the requirements of the LSTM. Further, the reshaped data frame was used to make the predictions with a loaded LSTM model and predictions were saved in the predictions variable. Further, the predictions of the data frame were append in the data frame with the column name 'Traffic Jam'. The date column that was set as an index in the previous function was reset after making the predictions.

Further few filtrations were applied to the data frame to fulfil your requirements. Firstly, the samples that have the time part in > 7 pm were fetched. Secondly, the samples that have the time part in the date column in < 19 pm were filtered. The filtered data frame was further used to filter the days. In the DateTime library, 0 represents Monday and 7 represents Sunday. So, the dates that have the day index from 0 to 5 were filtered to retain the records between Monday – Friday. Lastly, the predictions result after applying filtrations were passed to the main file.

Main update 2 File

```python
import pandas as pd
import streamlit as st
from functions import main

st.set_page_config(page_title="Traffic JAM Prediction", page_icon=":guardsman:", layout="centered")
st.title("Traffic Congestion Prediction")

options = ["Breda--Tilburg", "Tilburg--Breda"]
selected_option = st.selectbox("Choose an option:", options)

df = main(0, 1) if options.index(selected_option) == 0 else main(1, 0)
dates_list = list(df.Date.unique())
```

This is the main file of the streamlit that will run to show the GUI. Here I show the first 12 lines firstly to explain to you. The above code snippet first imports the required libraries and the main function from the function.py file. Further, it set the page configurations and page title. Further, it prepares a list for 2 routes of our study and shows them as a dropdown list. After getting the route from the user it passed the starting state and ending state to the main function and received the predictions as the methodology explained above. Further, it extracts the unique dates in line # 12.

```python
display_flag = False
button_dates = dates_list
print(button_dates)

for i in range(0, 4):
    for col in st.columns(4):
        with col:
            date = button_dates.pop(0) if len(button_dates) > 0 else None
            if date == None:
                break
            button = st.button(str(date))
            if button:
                df1 = df[df['Date'] == date]
                display_flag = True

if display_flag:
    st.write('Best Time to drive')
    st.dataframe(df1[['Time', 'Traffic Jam']])
```

Further few variables were used. The flag variable was set to False that used to show the results. By default, no results will show even the user clicks on the date button. Secondly, the dates-list was copied into the button-dates variable. Further a loop was iterated that display the button with the value as date. A condition was implemented in the loop that check either the button is click or not. If the button is clicked, then it filtered the samples of specific date and change the flag value to TRUE. The true flag value was checked lastly and display the data frame (last 3 lines).

Query – Answers

You said in the explanation that the model is 70% train and 30 test. I see here test size 0.2 doest that mean that its 80% train and 20 test.

Yes, you are right. We mistakenly write the 70% and 30% split but it was an 80% and 20% split for training and testing data.

what does the 50 next to lstm mean.

The 50 in LSTM represents the number of previous samples LSTM will use. There is no hard and fast rule for this. We set this according to the complexity of our problem.

the last thing why did you chose for 10 epoch with a batch size of 32

we set the batch size according to our computational resources. 32 batch size means the training set will divide into small chunks of length 32. If the size of the batch increases, then more samples will load in the memory and more times the weights will update. When all the chunks passed from the model once, we called it 1 epoch. So, the 10 epochs mean that all chunks will pass from the model 10 times. In other words, all the samples will pass from the model 10 times. There is no relation of batch size and epochs in NLP. However, you can ask why you stop the model on 10 epochs rather than the training on 100 or 1000 epochs. The answer is, we did but the model bent towards overfitting. That's why just train the model on 10 epochs and evaluate the significant results.