Data Prepping

```
#importing relevant packages

import os
import datetime
import IPython
import IPython.display
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, LeakyReLU
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, LearningRateScheduler
from sklearn.metrics import mean_squared_error
from tensorflow.keras import regularizers
from keras.regularizers import l2

mpl.rcParams['figure.figsize'] = (8, 6)
mpl.rcParams['axes.grid'] = False
```

```
#bringing in dataset

df = pd.read_csv('/content/drive/MyDrive/Data 4700/CityOfClevelandFinalData.csv')
```

```
#getting a sense of the unique census tracts

df['DW_Tract2020'].unique()
```

```
array([39035101101, 39035101102, 39035101300, 39035101400, 39035101501,
       39035101603, 39035101700, 39035101800, 39035101901, 39035102101,
       39035102102, 39035102200, 39035102300, 39035102401, 39035102402,
       39035102700, 39035102800, 39035102900, 39035103300, 39035103500,
       39035103602, 39035103800, 39035104400, 39035104800, 39035105100,
       39035105300, 39035105400, 39035105500, 39035105602, 39035105700,
       39035105900, 39035106100, 39035106200, 39035106500, 39035106600,
       39035106800, 39035106900, 39035107000, 39035107101, 39035107701,
       39035107802, 39035108201, 39035108301, 39035108400, 39035108701,
       39035109301, 39035109701, 39035109801, 39035110901, 39035111202,
       39035111401, 39035111700, 39035112100, 39035112200, 39035112301,
       39035114501, 39035114600, 39035115400, 39035115700, 39035115800,
       39035115900, 39035116300, 39035116400, 39035116500, 39035116600,
       39035116700, 39035116800, 39035116900, 39035117101, 39035117102,
       39035117201, 39035117300, 39035117400, 39035117500, 39035117600,
       39035117700, 39035117800, 39035117900, 39035118101, 39035118200,
       39035118301, 39035118602, 39035118800, 39035118900, 39035119401,
       39035119402, 39035119501, 39035119502, 39035119600, 39035119701,
       39035119702, 39035119800, 39035119900, 39035120200, 39035120400,
       39035120500, 39035120600, 39035120701, 39035120702, 39035120801,
       39035120802, 39035121100, 39035121200, 39035121300, 39035121401,
       39035121403, 39035121500, 39035121700, 39035121800, 39035121900,
       39035122100, 39035122200, 39035122300, 39035123100, 39035123200,
       39035123400, 39035123501, 39035123502, 39035123601, 39035123602,
       39035123603, 39035123700, 39035123800, 39035123900, 39035124100,
       39035124201, 39035124202, 39035124300, 39035124500, 39035124600,
       39035126100, 39035127501, 39035196400])
```

```
#getting a sense of the data as a whole

df.head()
```

	DW_Tract2020	TotalPermits	TotalJobValue	Housing_Permits	Businesses_Permits	Institutional Care_Permits	Food_Permits	Recreatio
0	39035101101	18.0	133923.0	6.0	1.0	0.0	0.0	
1	39035101101	17.0	120500.0	6.0	1.0	0.0	0.0	
2	39035101101	21.0	158680.0	7.0	1.0	0.0	0.0	
3	39035101101	19.0	106480.0	6.0	1.0	0.0	0.0	
4	39035101101	27.0	136588.0	6.0	1.0	0.0	0.0	

```
#getting the summary statistics for each variable

df.describe().transpose()
```

	count	mean	std	min	25%	50%	75%	max
DW_Tract2020	13699.0	3.903511e+10	1.054675e+04	3.903510e+10	3.903511e+10	3.903512e+10	3.903512e+10	3.903520e+10
TotalPermits	13699.0	3.717549e+01	2.768295e+01	0.000000e+00	2.200000e+01	3.200000e+01	4.500000e+01	4.120000e+02
TotalJobValue	13699.0	2.704331e+06	1.671471e+07	0.000000e+00	1.625115e+05	3.043561e+05	8.487080e+05	6.116884e+08
Housing_Permits	13699.0	1.234842e+01	8.448646e+00	0.000000e+00	6.000000e+00	1.100000e+01	1.700000e+01	9.500000e+01
Businesses_Permits	13699.0	8.104971e-01	2.593781e+00	0.000000e+00	0.000000e+00	0.000000e+00	1.000000e+00	5.800000e+01
Institutional Care_Permits	13699.0	4.343383e-02	2.598837e-01	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	5.000000e+00
Food_Permits	13699.0	1.521279e-01	6.304766e-01	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	1.300000e+01
Recreation_Permits	13699.0	1.427841e-01	6.283743e-01	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	1.200000e+01
Educational_Permits	13699.0	9.117454e-02	3.290486e-01	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	4.000000e+00
Hazard_Permits	13699.0	3.722899e-03	6.090411e-02	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	1.000000e+00
TotalDemolitionPermits	13699.0	2.365355e+00	3.707261e+00	0.000000e+00	0.000000e+00	1.000000e+00	3.000000e+00	4.000000e+01
TotalDemolitionJobValue	13699.0	3.175049e+04	1.167922e+05	0.000000e+00	0.000000e+00	8.050000e+03	3.192250e+04	4.810000e+06
ResidentialPermits	13699.0	2.149719e+00	3.501329e+00	0.000000e+00	0.000000e+00	1.000000e+00	3.000000e+00	4.000000e+01
CommercialPermits	13699.0	2.156362e-01	6.035019e-01	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	1.000000e+01
Total_per_1000	13699.0	1.831516e+01	9.676771e+00	1.575727e+00	1.172679e+01	1.683683e+01	2.269130e+01	9.967799e+01
Violent_per_1000	13699.0	6.336392e+00	3.626713e+00	1.274264e-01	3.754330e+00	5.903865e+00	8.235731e+00	3.769781e+01
Nonviolent_per_1000	13699.0	1.147638e+01	6.463939e+00	1.125676e+00	7.339049e+00	1.016960e+01	1.381649e+01	8.805217e+01
Vice_per_1000	13699.0	5.023892e-01	4.992408e-01	0.000000e+00	1.680143e-01	3.638329e-01	6.676443e-01	4.755355e+00
OtherPermits	13699.0	2.350909e+01	2.191796e+01	0.000000e+00	1.300000e+01	1.900000e+01	2.700000e+01	3.450000e+02

```
#dropping columns that will not be used as features

df = df.drop(columns = ['ResidentialPermits', 'CommercialPermits'])
```

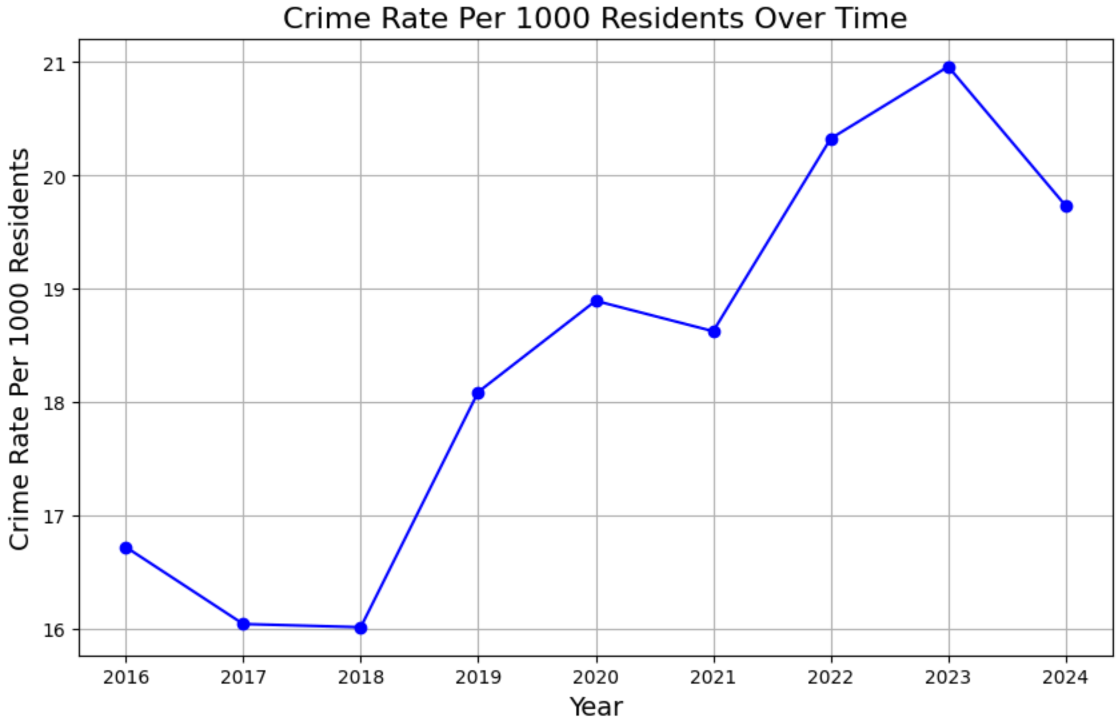
```
#getting data types

print(df.dtypes)
```

```
DW_Tract2020                   int64
TotalPermits                 float64
TotalJobValue                float64
Housing_Permits              float64
Businesses_Permits           float64
Institutional Care_Permits   float64
Food_Permits                 float64
Recreation_Permits           float64
Educational_Permits          float64
Hazard_Permits               float64
TotalDemolitionPermits       float64
TotalDemolitionJobValue      float64
Total_per_1000               float64
Violent_per_1000             float64
Nonviolent_per_1000          float64
Vice_per_1000                float64
Window                        object
OtherPermits                 float64
dtype: object
```

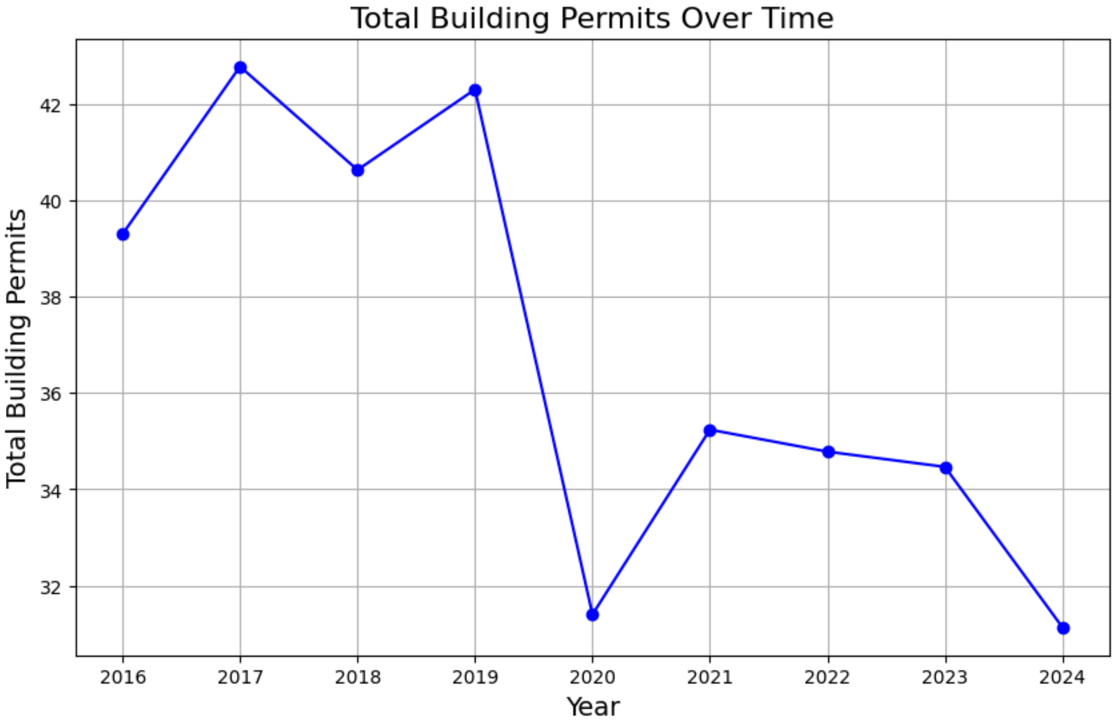
Basic Plots

```
#making a year column for plots

plotdf = df.copy(deep=True)

plotdf[['Date', 'Date2']] = plotdf['Window'].str.split(' to ', expand=True)
plotdf['Date'] = pd.to_datetime(plotdf['Date'])
plotdf['Year'] = plotdf['Date'].dt.year
```

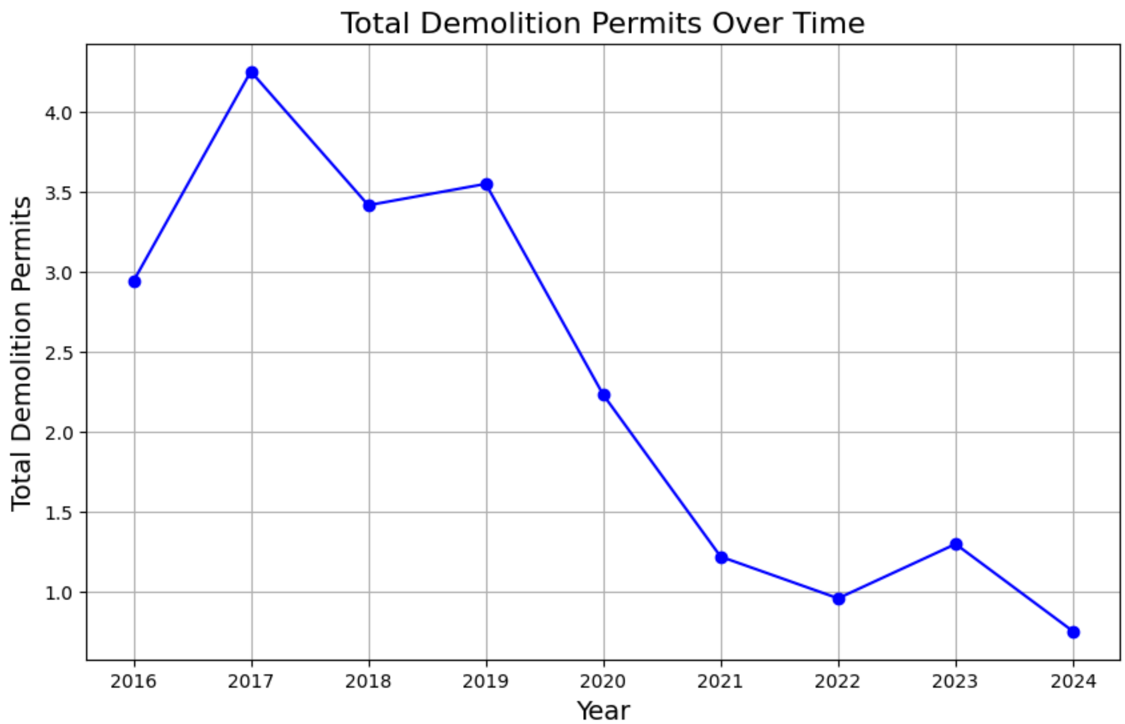
```
#plotting crime rate over year

df_avg = plotdf.groupby('Year')['Total_per_1000'].mean().reset_index()

plt.figure(figsize=(10, 6))
plt.plot(df_avg['Year'], df_avg['Total_per_1000'], marker='o', linestyle='-', color='b')

# Add titles and labels
plt.title('Crime Rate Per 1000 Residents Over Time', fontsize=16)
plt.xlabel('Year', fontsize=14)
plt.ylabel('Crime Rate Per 1000 Residents', fontsize=14)

# Show grid
plt.grid(True)

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




```
#plotting building permits over time

df_avg2 = plotdf.groupby('Year')['TotalPermits'].mean().reset_index()

plt.figure(figsize=(10, 6))
plt.plot(df_avg2['Year'], df_avg2['TotalPermits'], marker='o', linestyle='-', color='b')

# Add titles and labels
plt.title('Total Building Permits Over Time', fontsize=16)
plt.xlabel('Year', fontsize=14)
plt.ylabel('Total Building Permits', fontsize=14)

# Show grid
plt.grid(True)

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




```
#plotting demolition permits over time

df_avg3 = plotdf.groupby('Year')['TotalDemolitionPermits'].mean().reset_index()

plt.figure(figsize=(10, 6))
plt.plot(df_avg3['Year'], df_avg3['TotalDemolitionPermits'], marker='o', linestyle='-', color='b')

# Add titles and labels
plt.title('Total Demolition Permits Over Time', fontsize=16)
plt.xlabel('Year', fontsize=14)
plt.ylabel('Total Demolition Permits', fontsize=14)

# Show grid
plt.grid(True)

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



Baseline

```
df_baseline = df.copy(deep=True)
```

```
df_baseline[['Window_Start', 'Window_End']] = df_baseline['Window'].str.split(' to ', expand=True)
df_baseline['Window_Start'] = pd.to_datetime(df_baseline['Window_Start'])
df_baseline['Window_End'] = pd.to_datetime(df_baseline['Window_End'])
```

```
df_baseline['Year'] = df_baseline['Window_Start'].dt.year
```

```
average_crimes_year = df_baseline.groupby('Year')['Total_per_1000'].mean().reset_index()
average_crimes_year
```

	Year	Total_per_1000
0	2016	16.725217
1	2017	16.046384
2	2018	16.018394
3	2019	18.092298
4	2020	18.895973
5	2021	18.627781
6	2022	20.326012
7	2023	20.961436
8	2024	19.734172

```
df_baseline = pd.merge(df_baseline, average_crimes_year, on='Year', how='left')
```

```
df_baseline['mae'] = (df_baseline['Total_per_1000_x'] - df_baseline['Total_per_1000_y']).abs()
```

```
df_baseline['mae'].mean()
```

```
6.867110119094379
```

```
df_baseline.groupby('Year')['mae'].mean().reset_index()
```

	Year	mae
0	2016	5.966789
1	2017	6.122295
2	2018	5.711115
3	2019	6.729089
4	2020	6.957515
5	2021	6.745495
6	2022	7.593113
7	2023	8.238206
8	2024	8.364132

Splitting Data

```
#copying the data to a new dataset for features

features_df = df
```

```
#splitting the data by unique census tracts (70% train, 20% validation, 10% test)

unique_tract_numbers = features_df['DW_Tract2020'].unique()
sampled_tract_numbers = pd.Series(unique_tract_numbers).sample(frac=0.7, random_state=42)

train_df1 = features_df[features_df['DW_Tract2020'].isin(sampled_tract_numbers)]

remaining_df = features_df[~features_df['DW_Tract2020'].isin(sampled_tract_numbers)]

unique_tract_numbers2 = remaining_df['DW_Tract2020'].unique()
sampled_tract_numbers2 = pd.Series(unique_tract_numbers2).sample(frac=2/3, random_state=42)

val_df1 = features_df[features_df['DW_Tract2020'].isin(sampled_tract_numbers2)]

sampled_tract_numbers_combined = set(sampled_tract_numbers).union(set(sampled_tract_numbers2))
test_df1 = features_df[~features_df['DW_Tract2020'].isin(sampled_tract_numbers_combined)]
```

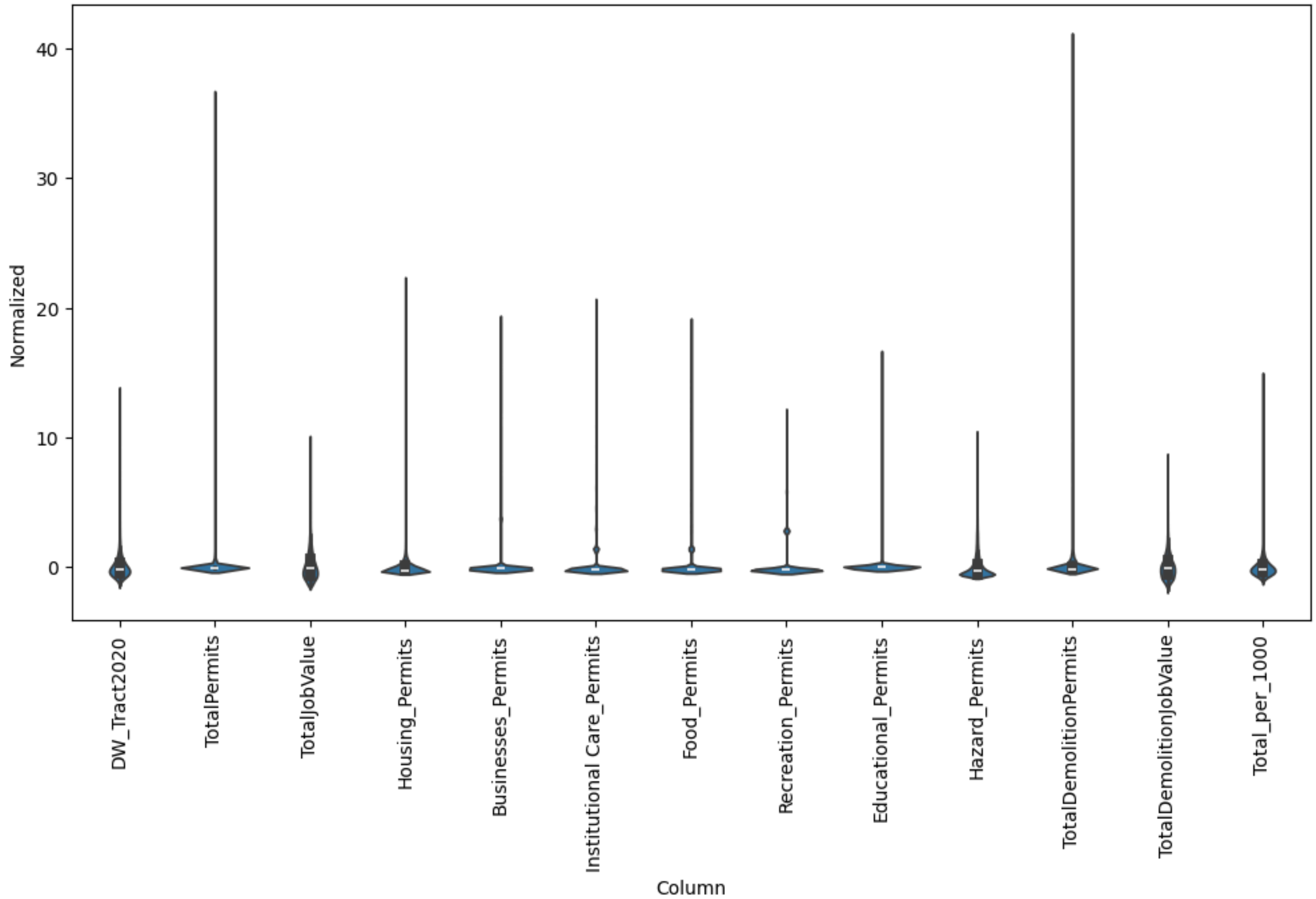
```
#creating temporary datasets for normalization and dropping columns that should not be normalized

train_df = train_df1.drop(columns = ['Violent_per_1000', 'Nonviolent_per_1000', 'Vice_per_1000'])
val_df = val_df1.drop(columns = ['Violent_per_1000', 'Nonviolent_per_1000', 'Vice_per_1000'])
test_df = test_df1.drop(columns = ['Violent_per_1000', 'Nonviolent_per_1000', 'Vice_per_1000'])
```

Normalization

```
features = df.copy(deep=True)
features = features.drop(columns=['DW_Tract2020', 'Violent_per_1000', 'Nonviolent_per_1000', 'Vice_per_1000', 'Window']
mean = features.mean()
std = features.std()
```

```
df_std = (features - mean) / std
df_std = df_std.melt(var_name='Column', value_name='Normalized')
plt.figure(figsize=(12, 6))
ax = sns.violinplot(x='Column', y='Normalized', data=df_std)
_ = ax.set_xticklabels(df.keys(), rotation=90)
```

```
<ipython-input-24-1a6649e1d958>:5: UserWarning: set_ticklabels() should only be used with a fixed number of ticks, i.e.
  _ = ax.set_xticklabels(df.keys(), rotation=90)
```




```
columns_to_normalize = ['TotalPermits', 'TotalJobValue', 'Housing_Permits', 'Businesses_Permits',
                        'Institutional Care_Permits', 'Food_Permits', 'Recreation_Permits',
                        'Educational_Permits', 'Hazard_Permits', 'TotalDemolitionPermits',
                        'TotalDemolitionJobValue', 'Total_per_1000', 'OtherPermits']
```

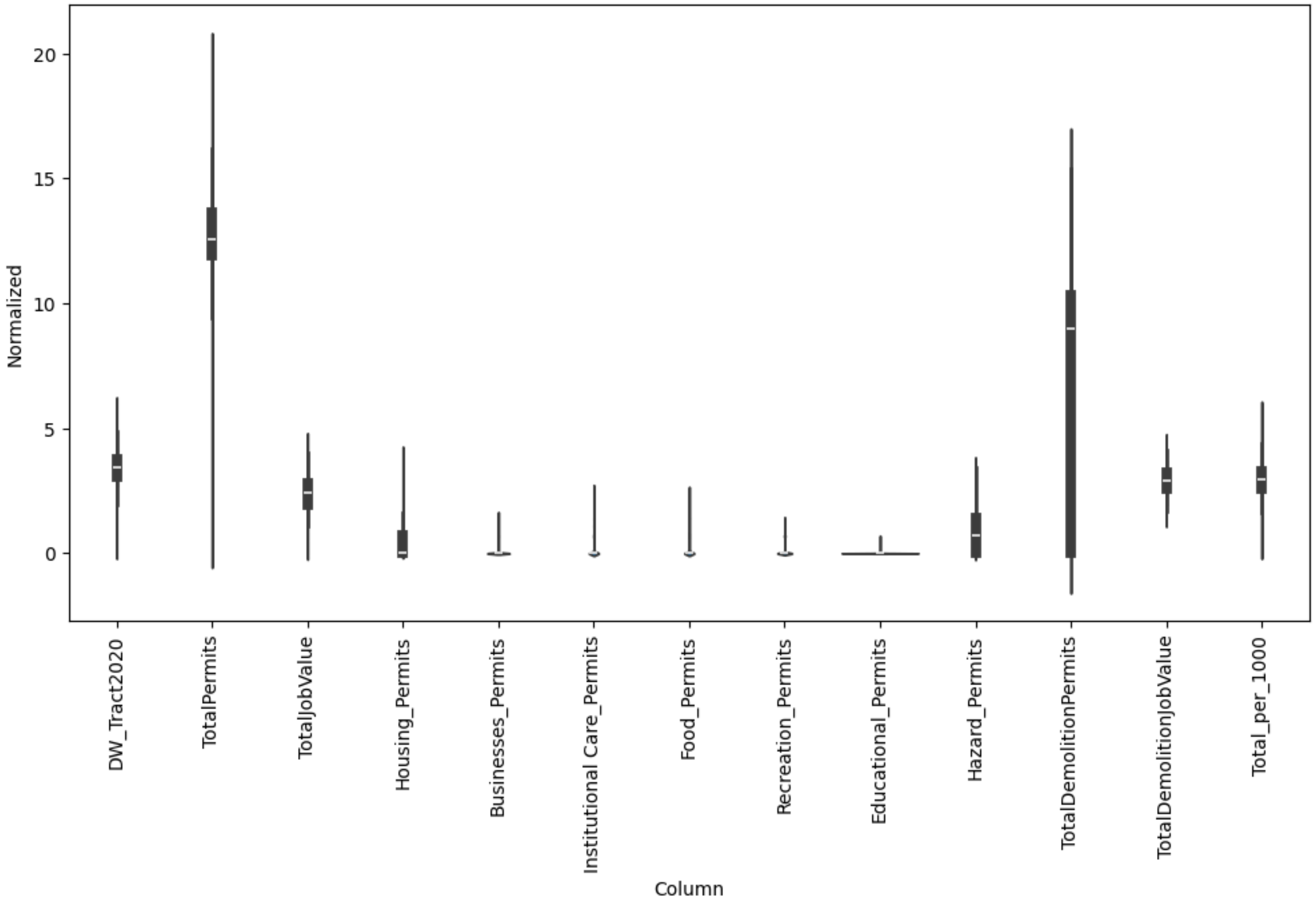
```
train_df[columns_to_normalize] = train_df[columns_to_normalize] + 1
val_df[columns_to_normalize] = val_df[columns_to_normalize] + 1
test_df[columns_to_normalize] = test_df[columns_to_normalize] + 1
```

```
train_df[columns_to_normalize] = np.log(train_df[columns_to_normalize])
val_df[columns_to_normalize] = np.log(val_df[columns_to_normalize])
test_df[columns_to_normalize] = np.log(test_df[columns_to_normalize])
```

```
features2 = train_df.copy(deep=True)
features2 = features2.drop(columns=['DW_Tract2020','Window'])

features3 = val_df.copy(deep=True)
features3 = features3.drop(columns=['DW_Tract2020','Window'])

features4 = test_df.copy(deep=True)
features4 = features4.drop(columns=['DW_Tract2020','Window'])
```

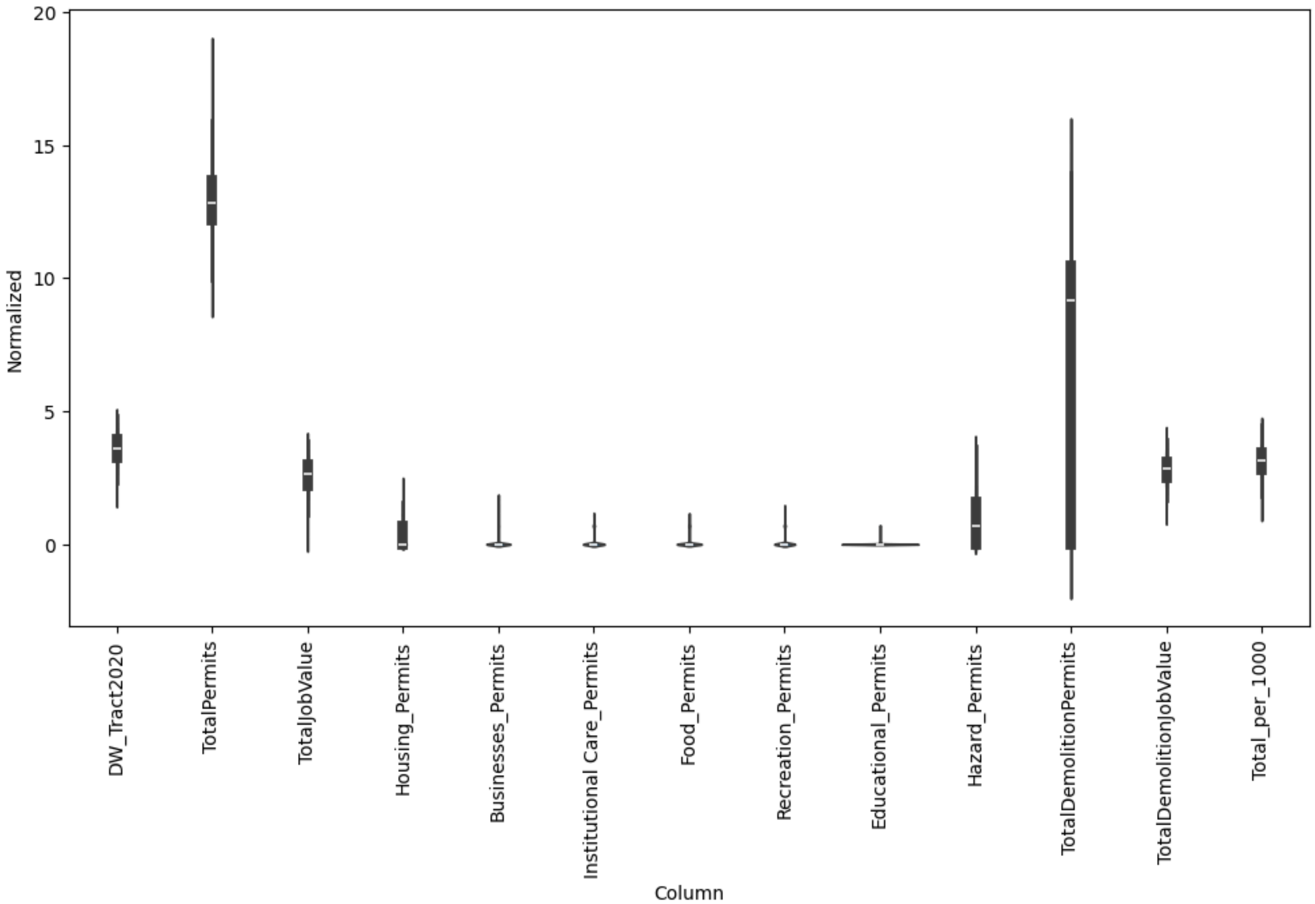
```
features2 = features2.melt(var_name='Column', value_name='Normalized')
plt.figure(figsize=(12, 6))
ax = sns.violinplot(x='Column', y='Normalized', data=features2)
_ = ax.set_xticklabels(df.keys(), rotation=90)
```

```
<ipython-input-29-97e73f084626>:4: UserWarning: set_ticklabels() should only be used with a fixed number of ticks, i.e.
  _ = ax.set_xticklabels(df.keys(), rotation=90)
```



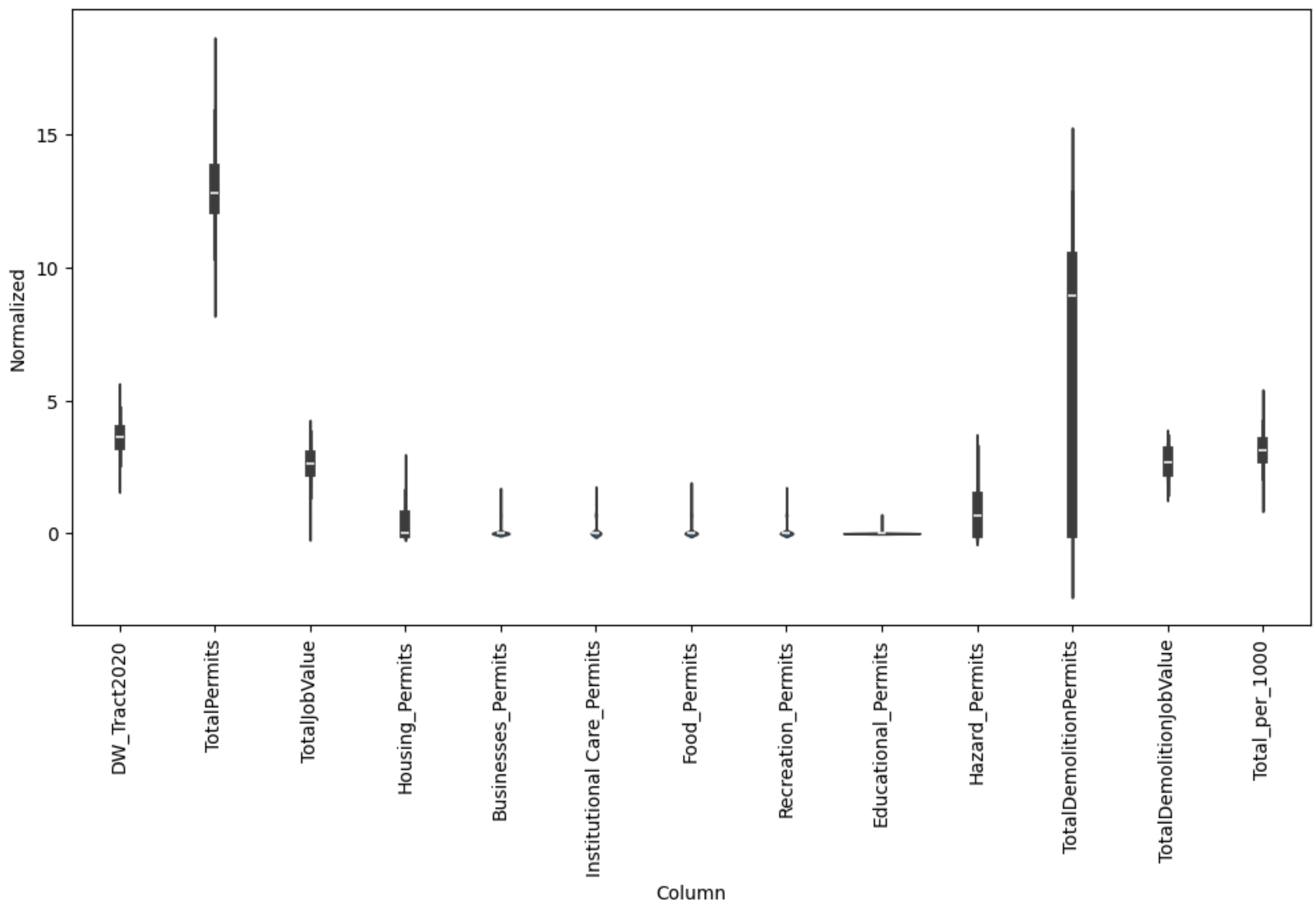

```
features3 = features3.melt(var_name='Column', value_name='Normalized')
plt.figure(figsize=(12, 6))
ax = sns.violinplot(x='Column', y='Normalized', data=features3)
_ = ax.set_xticklabels(df.keys(), rotation=90)
```

```
<ipython-input-30-b679bc86bea6>:4: UserWarning: set_ticklabels() should only be used with a fixed number of ticks, i.e.
  _ = ax.set_xticklabels(df.keys(), rotation=90)
```




```
features4 = features4.melt(var_name='Column', value_name='Normalized')
plt.figure(figsize=(12, 6))
ax = sns.violinplot(x='Column', y='Normalized', data=features4)
_ = ax.set_xticklabels(df.keys(), rotation=90)
```

```
<ipython-input-31-ad06e687cc78>:4: UserWarning: set_ticklabels() should only be used with a fixed number of ticks, i.e.
  _ = ax.set_xticklabels(df.keys(), rotation=90)
```



Windowing

```
#creating columns for the start and end of the time window in the train, validation, and test dataframes

train_df[['Window_Start', 'Window_End']] = train_df['Window'].str.split(' to ', expand=True)
val_df[['Window_Start', 'Window_End']] = val_df['Window'].str.split(' to ', expand=True)
test_df[['Window_Start', 'Window_End']] = test_df['Window'].str.split(' to ', expand=True)

train_df['Window_Start'] = pd.to_datetime(train_df['Window_Start'])
train_df['Window_End'] = pd.to_datetime(train_df['Window_End'])

val_df['Window_Start'] = pd.to_datetime(val_df['Window_Start'])
val_df['Window_End'] = pd.to_datetime(val_df['Window_End'])

test_df['Window_Start'] = pd.to_datetime(test_df['Window_Start'])
test_df['Window_End'] = pd.to_datetime(test_df['Window_End'])

train_df = train_df.sort_values(by=['DW_Tract2020', 'Window_Start'])
val_df = val_df.sort_values(by=['DW_Tract2020', 'Window_Start'])
test_df = test_df.sort_values(by=['DW_Tract2020', 'Window_Start'])
```

```
#creating windows for each census tract

class WindowGenerator():
    def __init__(self, input_width, label_width, shift, df, label_columns=None):
        self.df = df
        self.input_width = input_width
        self.label_width = label_width
        self.shift = shift
        self.label_columns = label_columns

    def create_windowed_sequences(self):
        inputs = []
        labels = []
```

```
        self.columns_to_drop = ['DW_Tract2020', 'Window', 'Window_Start', 'Window_End']
        for tract in self.df['DW_Tract2020'].unique():
            tract_data = self.df[self.df['DW_Tract2020'] == tract]
            for i in range(len(tract_data) - self.input_width - self.shift):
                input_data = tract_data.iloc[i:i + self.input_width].drop(columns=self.columns_to_drop)
                label_data = tract_data.iloc[i + self.input_width + self.shift - self.label_width: i + self.input_widtl
                inputs.append(input_data.values)
                labels.append(label_data.values)
        return np.array(inputs), np.array(labels)
```

```
#defining parameters: looking at 12 months of data, predicting 1 month, predicting a year later

input_width = 6
label_width = 1
shift = 12

#creating the windows with these parameters for the train, validation, and test dataframes

window_train = WindowGenerator(input_width=input_width, label_width=label_width, shift=shift, df=train_df)
X_train, y_train = window_train.create_windowed_sequences()

window_val = WindowGenerator(input_width=input_width, label_width=label_width, shift=shift, df=val_df)
X_val, y_val = window_val.create_windowed_sequences()

window_test = WindowGenerator(input_width=input_width, label_width=label_width, shift=shift, df=test_df)
X_test, y_test = window_test.create_windowed_sequences()
```

```
#analzying the shape of the dataframes

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_val shape:", X_val.shape)
print("y_val shape:", y_val.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: (7905, 6, 13)
y_train shape: (7905, 1)
X_val shape: (2295, 6, 13)
y_val shape: (2295, 1)
X_test shape: (1105, 6, 13)
y_test shape: (1105, 1)
```

Building The Model

```
def build_lstm_model(input_shape):
    model = models.Sequential()

    #LSTM layer with 64 units and relu activation
    model.add(layers.Bidirectional(layers.LSTM(64, return_sequences=True, input_shape=input_shape, kernel_regularizer=

    #LSTM layer with 32 unites and relu activation
    model.add(layers.Bidirectional(layers.LSTM(32)))

    #Output layer - single output for the predicted crime rate (Total_per_1000)
    model.add(layers.Dense(1), activation = 'relu')

    #Model is compiled using adam optimizer for learning rate and mse for loss
    model.compile(optimizer='adam', loss='mean_squared_error')

    return model
```

```
#constructing the model

def create_model(input_shape, units=32, dropout_rate=0.3):
    model = Sequential()
    model.add(LSTM(units, activation='relu', input_shape=input_shape, return_sequences=True))
    model.add(Dropout(dropout_rate))
    model.add(LSTM(units, activation='relu'))
    model.add(Dropout(dropout_rate))
```

```
    model.add(Dense(1))
    model.compile(optimizer=Adam(), loss='mean_squared_error', metrics=['mae'])
    return model

#altering the learning rate

def lr_schedule(epoch):
    initial_lr = 0.001
    drop = 0.5
    epochs_drop = 10
    return initial_lr * (drop ** (epoch // epochs_drop))

#early stopping used to optimize epochs

early_stopping = EarlyStopping(monitor='val_loss', patience=4, restore_best_weights=True)
lr_scheduler = LearningRateScheduler(lr_schedule)

#training the model

input_shape = (X_train.shape[1], X_train.shape[2])
model = create_model(input_shape)
history = model.fit(X_train, y_train, epochs=20, batch_size=128, validation_data=(X_val, y_val), callbacks=[early_stopp
```

```
Epoch 1/20
/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/rnn.py:204: UserWarning: Do not pass an `input_shape`/`inpu
  super().__init__(**kwargs)
124/124 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - loss: 1.2609 - mae: 0.8637 - val_loss: 0.1407 - val_mae: 0.2847 - learning_ra
Epoch 2/20
124/124 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 0.4137 - mae: 0.5090 - val_loss: 0.0978 - val_mae: 0.2327 - learning_ra
Epoch 3/20
124/124 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 0.3509 - mae: 0.4668 - val_loss: 0.0959 - val_mae: 0.2316 - learning_r
Epoch 4/20
124/124 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 0.2896 - mae: 0.4272 - val_loss: 0.0899 - val_mae: 0.2275 - learning_ra
Epoch 5/20
124/124 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - loss: 0.2769 - mae: 0.4158 - val_loss: 0.0665 - val_mae: 0.1961 - learning_ra
Epoch 6/20
124/124 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 0.2505 - mae: 0.3963 - val_loss: 0.0685 - val_mae: 0.2004 - learning_ra
Epoch 7/20
124/124 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 0.2312 - mae: 0.3795 - val_loss: 0.0802 - val_mae: 0.2232 - learning_ra
Epoch 8/20
124/124 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 0.2204 - mae: 0.3697 - val_loss: 0.0728 - val_mae: 0.2115 - learning_ra
Epoch 9/20
124/124 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - loss: 0.2089 - mae: 0.3610 - val_loss: 0.0887 - val_mae: 0.2401 - learning_ra
```

```
#plotting training and validation loss

def plot_loss(history):
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

plot_loss(history)
```


```
#evaluating test loss

test_loss = model.evaluate(X_test, y_test)
```

```
35/35 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0652 - mae: 0.2009
```

Making Predictions

```
#getting predictions

predictions = model.predict(X_test)
```

```
35/35 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step
```

```
print('Shape of predictions:', predictions.flatten().shape)
print('Shape of relevant test_df subset:', test_df['Window_End'].iloc[input_width + shift:].shape)
```

```
Shape of predictions: (1105,)
Shape of relevant test_df subset: (1321,)
```

```
#creating dataframe for predictions

relevant_window_end = test_df['Window_End'].iloc[input_width + shift:input_width + shift + len(predictions)].reset_inde
relevant_test_data = test_df.iloc[input_width + shift:input_width + shift + len(predictions)].reset_index(drop=True)
predictions_df = relevant_test_data.copy()
predictions_df['Predicted_Total_per_1000'] = predictions.flatten()
predictions_df['Actual_Total_per_1000'] = test_df['Total_per_1000'].iloc[input_width + shift + 12:input_width + shift -
```

```
columns_to_unnormalize = ['TotalPermits', 'TotalJobValue', 'Housing_Permits', 'Businesses_Permits',
                          'Institutional Care_Permits', 'Food_Permits', 'Recreation_Permits',
                          'Educational_Permits', 'Hazard_Permits', 'TotalDemolitionPermits',
                          'TotalDemolitionJobValue', 'Total_per_1000', 'OtherPermits', 'Predicted_Total_per_1000', 'Actua
```

```
#undoing the normalization for the predicted and actual crime rate

predictions_df[columns_to_unnormalize] = np.exp(predictions_df[columns_to_unnormalize]) - 1
```

```
#analyzing the predictions

predictions_df.head(20)
```

	DW_Tract2020	TotalPermits	TotalJobValue	Housing_Permits	Businesses_Permits	Institutional Care_Permits	Food_Permits	Recreatic
0	39035103602	168.0	14145635.06	39.0	2.0	1.0	1.0	
1	39035103602	150.0	11676584.06	31.0	1.0	1.0	1.0	
2	39035103602	145.0	10447819.06	30.0	2.0	1.0	1.0	
3	39035103602	135.0	11950135.06	25.0	3.0	2.0	1.0	
4	39035103602	133.0	8912718.00	24.0	4.0	1.0	0.0	
5	39035103602	158.0	9988647.00	24.0	7.0	1.0	2.0	

Accuracy

```
predictions_df['mae'] = (predictions_df['Predicted_Total_per_1000'] - predictions_df['Actual_Total_per_1000']).abs()
```

```
predictions_df['mae'].mean()
```

6.118318780321306

Analzying Prediction Accuracy By Tract

```
mae_by_tract = predictions_df.groupby('DW_Tract2020')['mae'].mean().reset_index()

mae_by_tract
```

	DW_Tract2020	mae
0	39035103602	8.928127
1	39035106200	1.803884
2	39035110901	2.487461
3	39035115800	6.094560
4	39035117600	2.382560
5	39035118800	6.211372
6	39035119502	6.748293
7	39035119600	8.666774
8	39035121200	11.506537
9	39035122200	3.547053
10	39035123602	9.770503

Analzying Prediction Accuracy By Crime Rate Totals

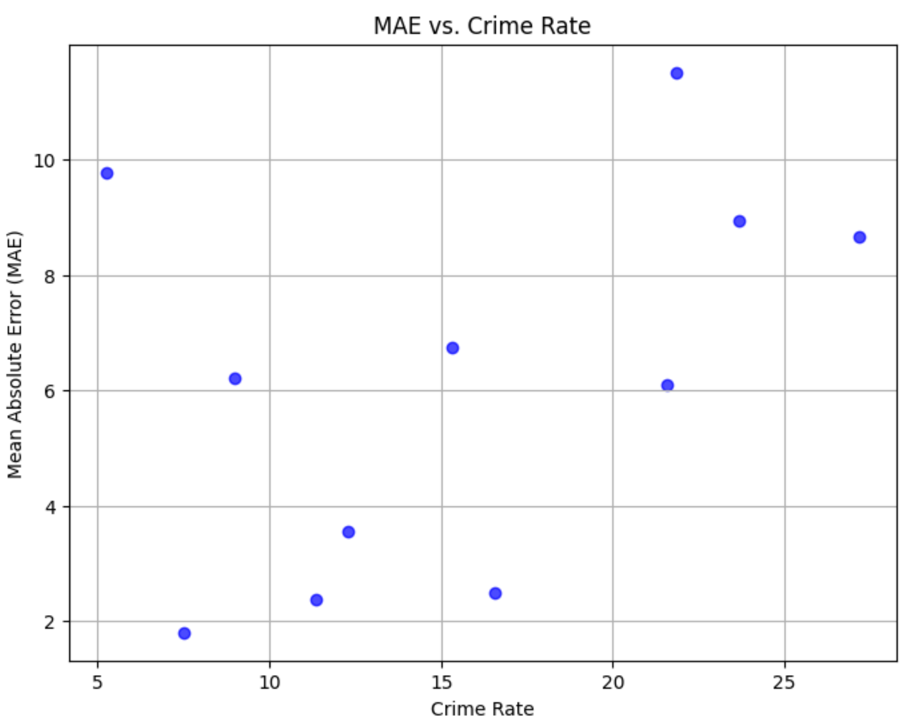
```
mae_crimes_by_tract = predictions_df.groupby('DW_Tract2020').agg(
    mae_mean=('mae', 'mean'),
    total_per_1000_mean=('Total_per_1000', 'mean')
```

```
).reset_index()

mae_crimes_by_tract
```

	DW_Tract2020	mae_mean	total_per_1000_mean
0	39035103602	8.928127	23.667090
1	39035106200	1.803884	7.503533
2	39035110901	2.487461	16.565453
3	39035115800	6.094560	21.592443
4	39035117600	2.382560	11.362488
5	39035118800	6.211372	8.982222
6	39035119502	6.748293	15.304366
7	39035119600	8.666774	27.159787
8	39035121200	11.506537	21.843370
9	39035122200	3.547053	12.291565
10	39035123602	9.770503	5.256676

				22.0	5.0	0.0	1.0
16	39035103602	105.0	9094877.68	22.0	4.0	0.0	1.0
				17.0	5.0	0.0	1.0
18	39035103602	109.0	9481195.08	18.0	7.0	0.0	1.0

```
plt.figure(figsize=(8, 6))
plt.scatter(mae_crimes_by_tract['total_per_1000_mean'], mae_crimes_by_tract['mae_mean'], color='b', alpha=0.7)  # Scatt
plt.title('MAE vs. Crime Rate')
plt.xlabel('Crime Rate')
plt.ylabel('Mean Absolute Error (MAE)')
plt.grid(True)

plt.show()
```

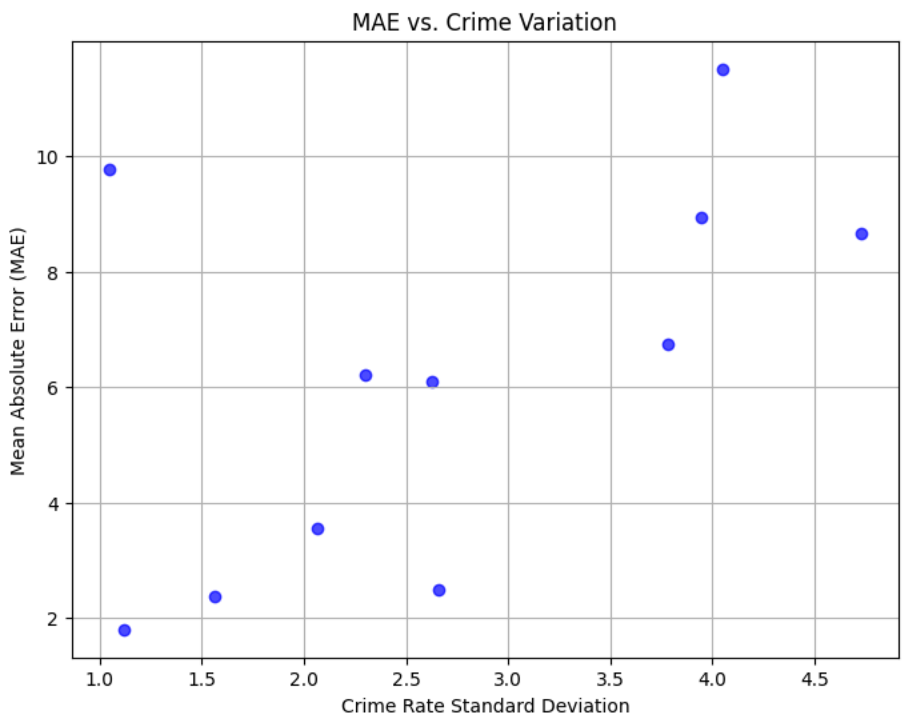


Analzying Prediction Accuracy By Crime Rate Variance

```
mae_crimes_by_tract_std = predictions_df.groupby('DW_Tract2020').agg(
    mae_mean=('mae', 'mean'),
    total_per_1000_std=('Total_per_1000', 'std')
).reset_index()

mae_crimes_by_tract_std
```

	DW_Tract2020	mae_mean	total_per_1000_std
0	39035103602	8.928127	3.945904
1	39035106200	1.803884	1.120428
2	39035110901	2.487461	2.658733
3	39035115800	6.094560	2.623536
4	39035117600	2.382560	1.565047
5	39035118800	6.211372	2.300071
6	39035119502	6.748293	3.780818
7	39035119600	8.666774	4.725869
8	39035121200	11.506537	4.045932
9	39035122200	3.547053	2.064885
10	39035123602	9.770503	1.048022

```
plt.figure(figsize=(8, 6))
plt.scatter(mae_crimes_by_tract_std['total_per_1000_std'], mae_crimes_by_tract_std['mae_mean'], color='b', alpha=0.7)
plt.title('MAE vs. Crime Variation')
plt.xlabel('Crime Rate Standard Deviation')
plt.ylabel('Mean Absolute Error (MAE)')
plt.grid(True)

plt.show()
```



Analzying Prediction Accuracy By Tract Population

```
populations = pd.read_csv('/content/drive/MyDrive/Data Capstone/CensusData.csv')
```

```
populations = populations[['Geographic Identifier - FIPS Code', 'Total Population']]
```

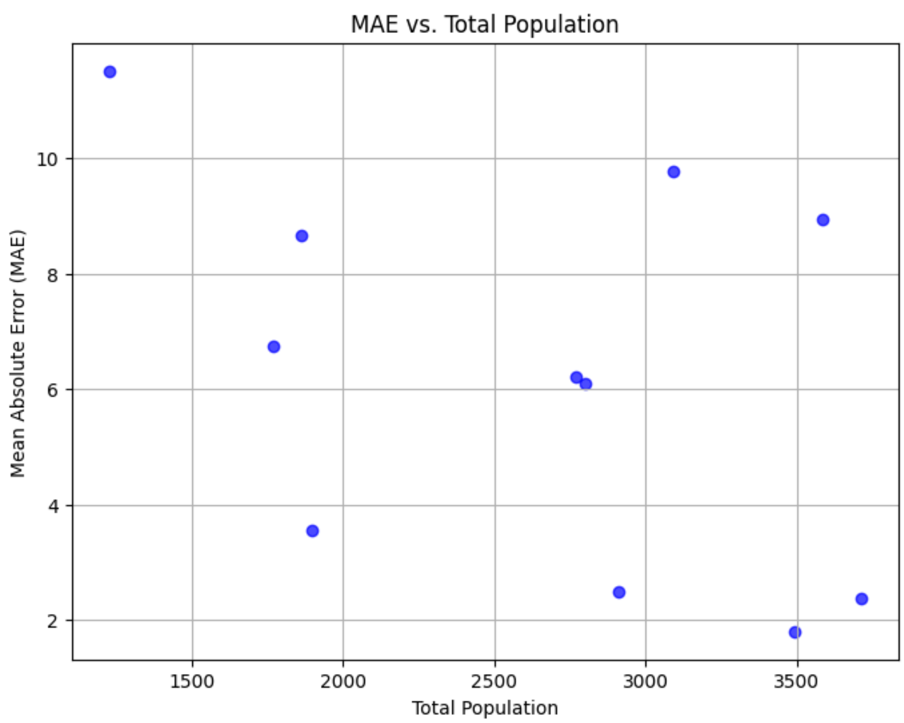
```
populations.rename(columns={'Geographic Identifier - FIPS Code': 'DW_Tract2020'}, inplace=True)
```

```
mae_populations = mae_by_tract.merge(populations, how='left', on='DW_Tract2020')
```

```
mae_populations
```

	DW_Tract2020	mae	Total Population
0	39035103602	8.928127	3584
1	39035106200	1.803884	3492
2	39035110901	2.487461	2912
3	39035115800	6.094560	2803
4	39035117600	2.382560	3712
5	39035118800	6.211372	2769
6	39035119502	6.748293	1770
7	39035119600	8.666774	1861
8	39035121200	11.506537	1229
9	39035122200	3.547053	1900
10	39035123602	9.770503	3093

```
plt.figure(figsize=(8, 6))
plt.scatter(mae_populations['Total Population'], mae_populations['mae'], color='b', alpha=0.7)
plt.title('MAE vs. Total Population')
plt.xlabel('Total Population')
plt.ylabel('Mean Absolute Error (MAE)')
plt.grid(True)

plt.show()
```



Analzying Prediction Accuracy By Year

```
predictions2024 = predictions_df[predictions_df['Window_Start'] > pd.Timestamp('2023-12-31')]
```

```
predictions2024 = predictions2024.copy(deep=True)

predictions2024['mae'] = (predictions2024['Predicted_Total_per_1000'] - predictions2024['Actual_Total_per_1000']).abs()

mae_by_tract2024 = predictions2024.groupby('DW_Tract2020')['mae'].mean().reset_index()

mae_by_tract2024
```

	DW_Tract2020	mae
0	39035103602	16.270704
1	39035106200	0.844541
2	39035110901	0.966176
3	39035115800	3.650623
4	39035117600	1.835010
5	39035118800	7.360134
6	39035119502	5.040640
7	39035119600	7.011194
8	39035121200	1.914435
9	39035122200	5.708568

```
predictions_df['Year'] = predictions_df['Window_Start'].dt.year
predictions_df.groupby('Year')['mae'].mean().reset_index()
```

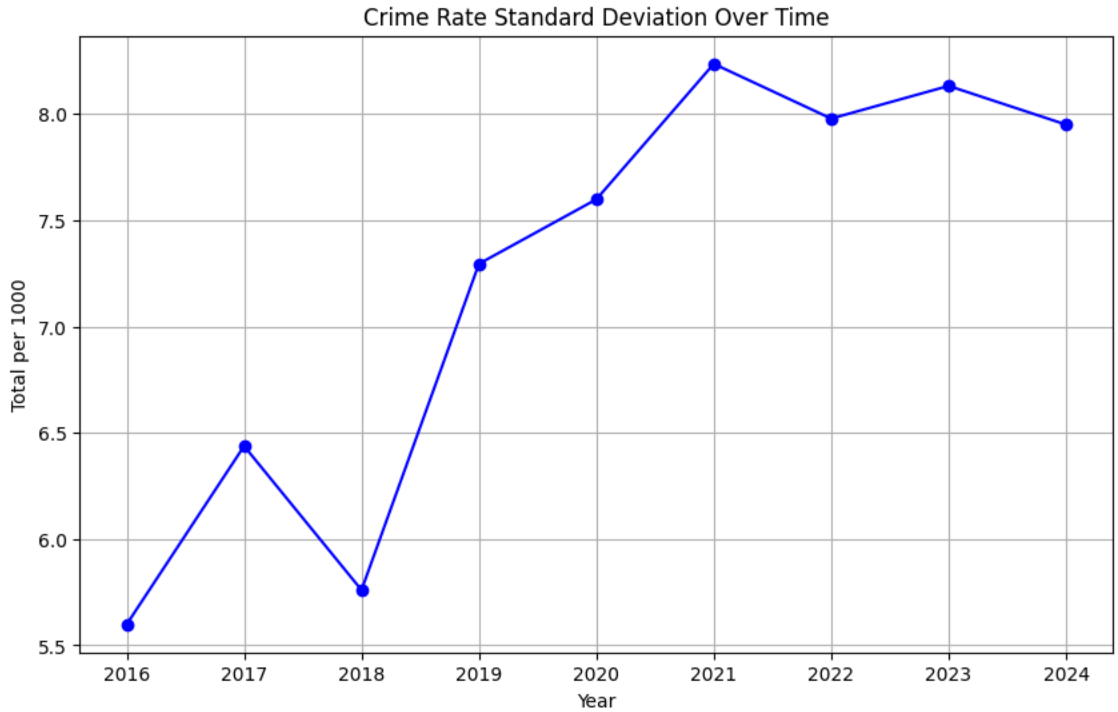
	Year	mae
0	2016	4.506544
1	2017	4.461900
2	2018	4.678185
3	2019	5.906784
4	2020	6.898809
5	2021	6.554076
6	2022	8.682986
7	2023	7.630951
8	2024	5.060202

Considerations

```
std_by_year = predictions_df.groupby('Year')['Total_per_1000'].std().reset_index()
std_by_year
```

	Year	Total_per_1000
0	2016	5.599041
1	2017	6.437787
2	2018	5.760473
3	2019	7.295039
4	2020	7.598408
5	2021	8.235666
6	2022	7.978183
7	2023	8.131943
8	2024	7.949896

```
plt.figure(figsize=(10, 6))
plt.plot(std_by_year['Year'], std_by_year['Total_per_1000'], marker='o', linestyle='-', color='b')

plt.title('Crime Rate Standard Deviation Over Time')
plt.xlabel('Year')
plt.ylabel('Total per 1000')

plt.grid(True)

plt.show()
```




```
yeardf = df.copy(deep=True)
yeardf[['Window_Start', 'Window_End']] = yeardf['Window'].str.split(' to ', expand=True)
yeardf['Window_Start'] = pd.to_datetime(yeardf['Window_Start'])
yeardf['Year'] = yeardf['Window_Start'].dt.year
```

```
std_by_year_df = yeardf.groupby('Year')['Total_per_1000'].std().reset_index()
std_by_year_df
```