# New York City Central Park Weather Prediction

Author: Jiayuan Shen, Ke Xu, Yan Liu

Course: DATA 602

Semester: Fall 2023

Professor: Mohammad T. Hajiaghayi

TA: Max Springer

# Background

For our concluding project, we have used the modeling methods to predict future temperatures. We have based our analysis on the GHCND dataset, obtained from the National Oceanic and Atmospheric Administration's website. This dataset offers extensive daily weather data, which has been crucial in fine-tuning and adjusting our models to improve their forecasting precision.

In our study, we focused on training and fine-tuning the machine learning models, each utilizing datasets that, while structurally similar, exhibit unique characteristics. A primary objective in this process was the identification and reduction of biases inherent within the machine learning algorithms. This was achieved by implementing strategies specifically designed to neutralize these biases, thereby enhancing the reliability and fairness of the models. This report is structured to elucidate the methodologies employed in this endeavor, detailing the steps taken to mitigate bias and the implications of these measures on the performance and integrity of the machine learning models.

# Introduction

In our initial methodology, we employed a neural network model comprising two hidden layers for the purpose of forecasting temperature (measured in Centigrade) at LaGuardia Airport, New York. This process encompassed the application of feature engineering techniques to the dataset to refine it for model training, followed by the use of the enhanced dataset for prediction of future meteorological conditions, specifically temperature, based on the identified features. Additionally, we developed a comprehensive dashboard designed to visually represent the interrelations among various features. This dashboard also served to illustrate the architecture of the neural network model and to facilitate the comparison of predicted meteorological outcomes against actual observed values. As a culminating step, our model was tasked with generating projected temperature values for a period extending 30 days beyond the current date.

# Web Scraping

1. Request the Web Services Token from the internet to gain access to NCDC CDO Web Services via email.
2. Follow the instructions to get the correct parameter.
    - To retrieve the required data, it is necessary to access the specified endpoint "/data?datasetid=YOUR_DATASETID," which can be augmented with various optional parameters to tailor the data retrieval process. Nevertheless, a key constraint to be mindful of is the limitation imposed on the number of results returned in each response. This necessitates a strategic approach in utilizing each call to the fullest extent, minimizing redundant requests and reducing the strain on server resources. Efficient usage of each call streamlines the data collection process and ensures mitigating the server's workload.

# Data Description and Cleaning

1. Combine previous data(2020-08-15 to 2023-11-01) and today's data(2023-11-02 to Today) into one dataframe.
2. In our preliminary analysis, we compared two datasets post data processing and identified 12 columns, including WT05 and RHMN, that were exclusive to the first dataset. Consequently, we combined these with the second dataset. This integration ensures data completeness and reliability, laying a solid foundation for further data processing.
3. To generate additional features, we appended three columns after the maximum temperature column (TMAX), each representing the maximum temperatures of the three days preceding the current date. This expansion increased our feature set from 24 to 58 columns, and then to 61 columns. In subsequent machine learning and data mining processes, the increased number of

features, signifying a richer information base, aids in a deeper understanding of the dataset and its patterns, facilitating the construction of more accurate models.

Our approach to handling missing values in the data involved the following methods:

1. For missing daily data, we calculated the average temperature for that day over the previous four years, using this average to replace any missing values.
2. The dataset included eight special weather indicators (WT01-WT08), as found in the metadata table. These indicators, representing the presence (1) or absence (0) of certain weather conditions, were assigned a value of 0 in place of NULL, indicating the non-occurrence of those weather conditions on given days.
3. For each rows, enlarged the data using features of three days before for columns (TAVG, TMAX, PRCP, SNOW, AWND, RHAV, RHMN, RHMX, ADPT, ASLP, ASTP);
4. Drop the missing data contained in those columns due to the shifting.

This dataset was split into 80% for training and 20% for validation for the next step of Model Training.

## Neural Network

We have also implemented a two-layer NN model based on Tensorflow keras. After deploying experiments between common ML models, we surprisingly found that a basic two-layer NN achieved high performance compared to other SOTA (State of the Art) ML models for this task. Generally speaking, NN has outstanding ability to handle data with time-series. For the reason above, we chose NN as our training model.

## Model Training

In model training, this dataset was split into 80% for training and 20% for validation.Our goal through this model is to predict and ascertain the average temperature for a target date. To enhance the

model's intuitiveness, we utilized the 'keras.utils' 'plot_model' function to visualize the model, producing the following image of our neural network model, The model plot we obtained is shown in Figure 1.

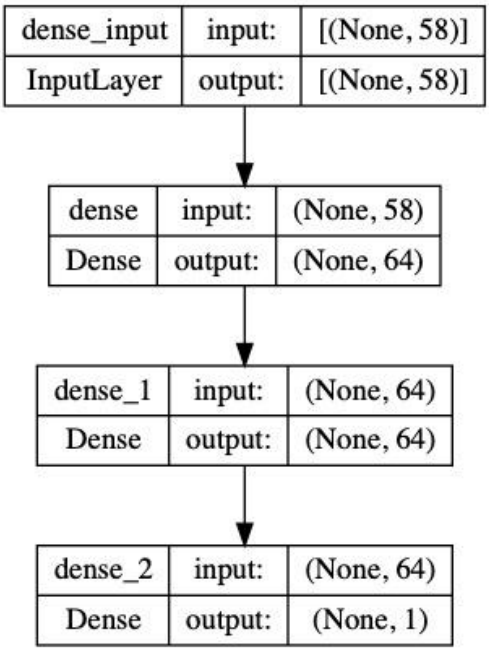


***Figure 1.*** *Model plot*

Initially, our objective was to forecast the subsequent day's weather conditions. Accordingly, we omitted columns that directly related to the current day's maximum, minimum, and average temperatures (TAVG, TMAX, TMIN). This refinement resulted in a modified training set comprising 58 features, with the subsequent day's average temperature as our prediction target.

Subsequently, we split the dataset into training and testing subsets. The training data was standardized using the ‘StandardScaler’. In designing our model architecture, we opted for the Exponential Linear Unit (ELU) activation function over the commonly used Rectified Linear Unit (ReLU), incorporating an L2 regularizer to curtail the risk of overfitting. Consistent with prevalent practices, we selected the Adam algorithm as our optimizer. The Mean Squared Error (MSE) was employed as the loss function, while the Mean Absolute Error (MAE) served as our metric for evaluation during testing.

Finally, to succinctly demonstrate the correlation between the actual and predicted average temperatures, we generated a scatter plot. We obtained this result: Mean Absolute Error: 0.9244971346682038, which indicates that our model is accurate. This visualization, as depicted in Figure 2 below, facilitates a clear comparison of our model's predictions against the observed data (see Figure 2).

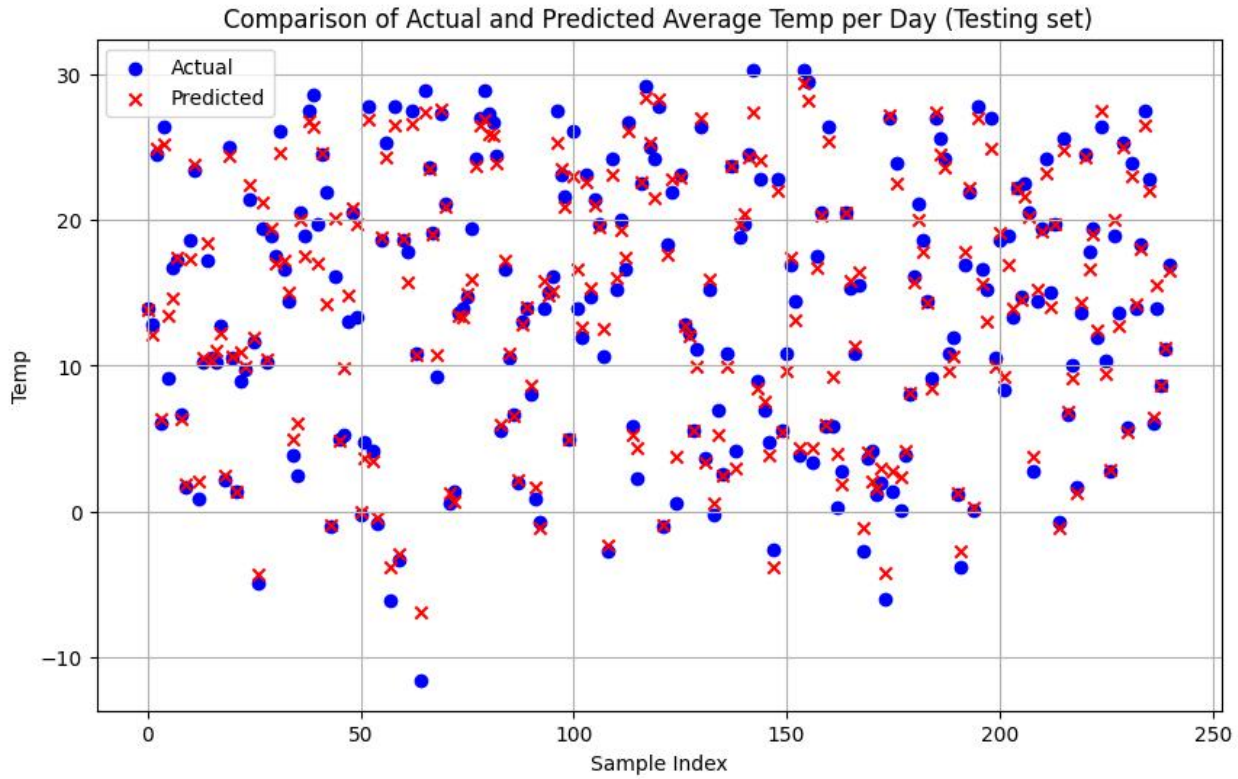


***Figure 2.*** *Comparison of actual and predicted average temperature*

Note: The result might vary due to the random initial weight setting of the Keras and today's date.

## Visualization Through Dashboard

We used dash from the dashboard to generate various types of graphs to facilitate the data analysis. Five graphs were generated to show the correlation between features, the model structure, the

predicted average temperature comparison and the weather forecast. In order to interact with these graphs, you need to run the colab notebook in your local computer. Click here to open the dashboard.

A cumulative line graph is an effective tool for illustrating the correlation between the target predictions and the feature values in our analysis. This graphical representation provides a clear visual of how these variables interact and underscores the rationale behind normalizing both the training and validation sets. Normalization, a crucial preprocessing step, ensures that feature values are comparable, enhancing the model's ability to interpret and learn from the data accurately. The cumulative line graph effectively demonstrates the impact of normalization on the relationship between the target and features, highlighting its importance in the context of predictive modeling(see Figure 3).

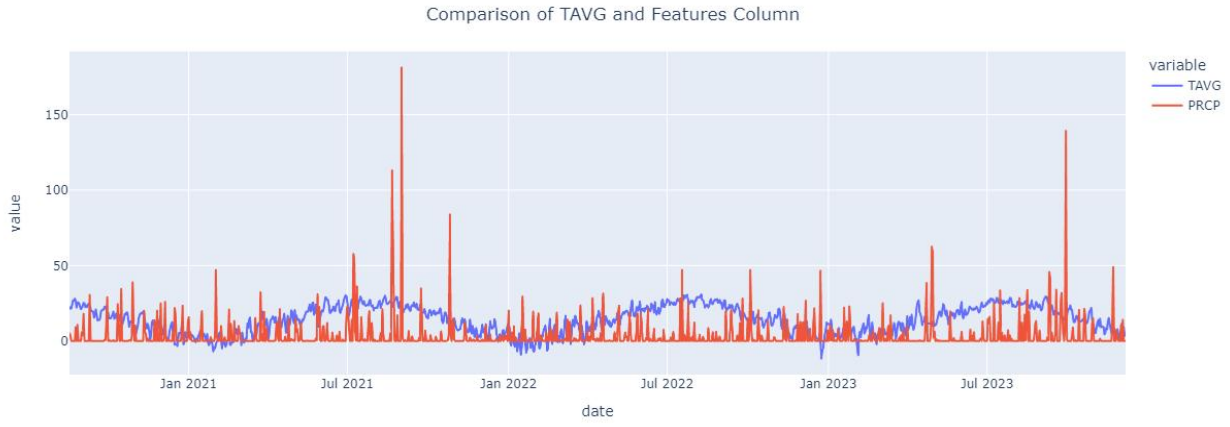


***Figure 3.*** *Comparison of TAVG and Features Column*

The second graph in our analysis is a heatmap designed to display the correlations among the various features vividly. This visualization technique is beneficial for identifying and understanding the strength and direction of relationships between different variables. Incorporating an app callback function enhances this tool, enabling real-time examination and direct interaction with the data through the front-end interface. This interactive element allows users to explore and analyze the correlation values dynamically, offering a more nuanced understanding of the data relationships as they engage with the heatmap (see Figure 4).

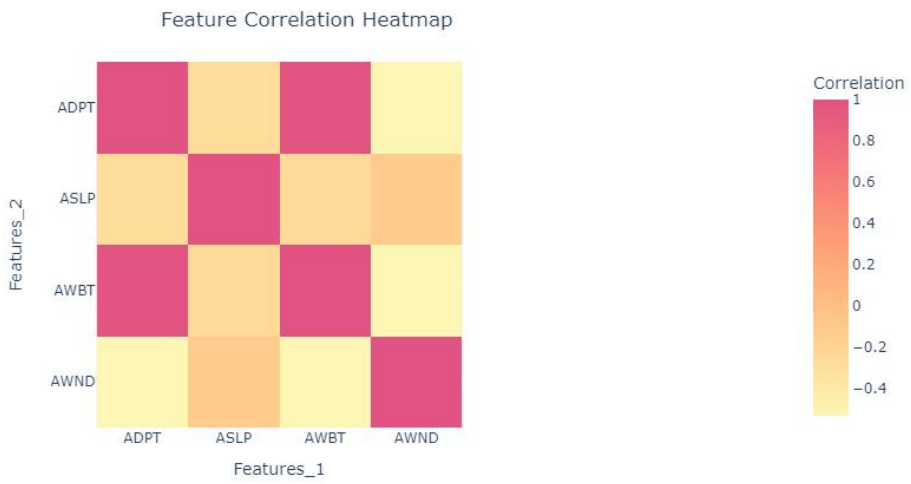


***Figure 4.*** *Feature Correlation Heatmap*

The third graph in our series is a structural representation of our machine learning model, specifically designed to facilitate reproducibility. This graphical depiction outlines the model's architecture, including the various layers and configurations. The primary purpose of this visual aid is to enable users, particularly those with more advanced insights or strategies, to modify the model easily. By providing a clear and detailed overview of the model structure, users are empowered to experiment with adding or removing layers to improve critical metrics such as the Mean Squared Error significantly. Additionally, these modifications could aid in mitigating data loss, thereby enhancing the overall effectiveness and accuracy of the model. This graph is crucial for collaborative and iterative model development and improvement (see Figure 1).

The fourth graph in our presentation is the key visual component, featuring two distinct lines: a blue line representing actual values and a red line depicting our model's predictions. This graph is designed with an interactive element – a sliding bar that allows users to navigate through the data and examine weather predictions for the upcoming 30 days. To enhance clarity and minimize the potential for misinterpretation or misleading conclusions, we have deliberately chosen to display only select points on the prediction line rather than the entire continuum of predicted values. This approach balances providing

comprehensive information and maintaining a clear, user-friendly visual representation that effectively communicates the critical insights derived from our model's forecasts (see Figure 5).

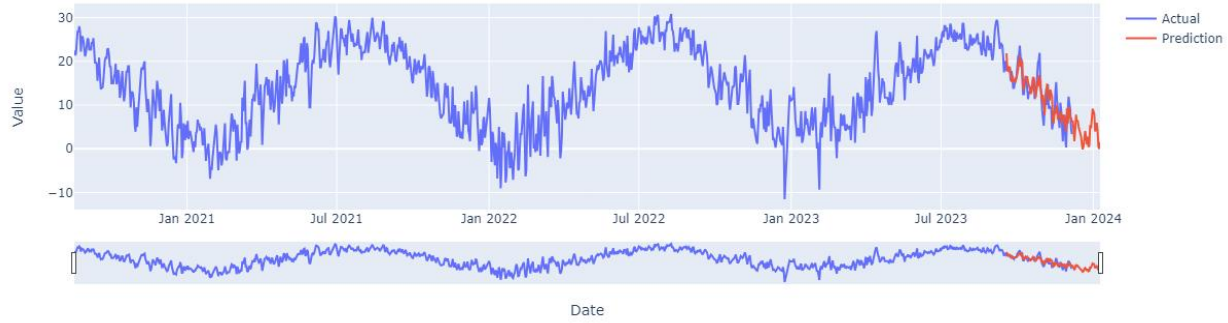


***Figure 5.*** *The dashboard of user interface of the weather forecast*

# Future Predictions and Limitations

Due to constraints in the availability of comprehensive weather data from open APIs, our model encountered limitations in accessing crucial data pertinent to temperature prediction. A notable issue was the inconsistent updating of cumulative features; while these features do get updated, the frequency varies, with some updating monthly or less frequently. To address this challenge, we adopted a compromise approach, utilizing the mean value of previous months and days as a predictive input. This strategy, however, introduced a trade-off in terms of accuracy, particularly in extreme weather conditions. Although our model achieved a mean absolute error of approximately 0.9, its performance was suboptimal in scenarios involving significant temperature fluctuations. Specifically, in instances of abrupt increases or decreases in temperature relative to preceding days, the model's predictions did not align as closely with actual conditions as desired.

# Conclusion

In our conclusion, we reflect on the entirety of our methodology, highlighting the diverse range of techniques and libraries in Python that we utilized to maximize our programming expertise. A key aspect of our approach involved diligently sourcing and aggregating data from multiple sites. This process was critical in ensuring comprehensive data coverage from each station, mitigating the risk of unforeseen data loss. We also endeavored to augment our model's performance by expanding the feature set, a strategy aimed at refining the predictive accuracy of our model.

Furthermore, our exploration into various models necessitated a patient and skilled approach, particularly in fine-tuning hyperparameters, including adjustments to the learning rate for gradient descent and the lambda for regularization, both crucial in preventing overfitting. In addition to these technical aspects, we also focused on incorporating anomaly variables to align our model more closely with traditional weather forecasting methodologies. This integration was intended to enhance the model's ability to predict weather patterns, thereby bolstering its overall effectiveness accurately.

Link to our Colab: Data_602_Final.ipynb

Link to our Metadata: DATA602 MetaData

Link to our Dataset: https://www.ncdc.noaa.gov/cdo-web/webservices/v2#data

# References


1. National Centers for Environmental Information (NCEI). (n.d.). *Climate Data Online: Web Services Documentation*. National Climatic Data Center. https://www.ncdc.noaa.gov/cdo-web/webservices/v2#gettingStarted
2. *National Centers for Environmental Information*. National Centers for Environmental Information (NCEI). (n.d.-a). https://www.ncei.noaa.gov/
3. *Pandas documentation#.* pandas documentation - pandas 2.1.4 documentation. (n.d.). https://pandas.pydata.org/docs/
4. *NumPy documentation#.* NumPy documentation - NumPy v1.26 Manual. (n.d.). https://numpy.org/doc/stable/
5. *Matplotlib.pyplot#.* matplotlib.pyplot - Matplotlib 3.5.3 documentation. (n.d.). https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html
6. *Plotly*. Getting started with plotly in Python. (n.d.). https://plotly.com/python/getting-started/
7. *Plotly*. Plotly express in Python. (n.d.). https://plotly.com/python/plotly-express/
8. *Seaborn*. An introduction to seaborn - seaborn 0.13.0 documentation. (n.d.). https://seaborn.pydata.org/tutorial/introduction
9. *Sklearn.model_selection.GRIDSEARCHCV*. scikit. (n.d.). https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
10. *Tensorflow 2 quickstart for beginners: Tensorflow Core*. TensorFlow. (n.d.). https://www.tensorflow.org/tutorials/quickstart/beginner
11. Chollet, F. (2020, April 12). *Keras Documentation: The sequential model*. Keras. https://keras.io/guides/sequential_model/
12. Chollet, F. (n.d.). *Keras documentation: Dense layer*. Keras. https://keras.io/api/layers/core_layers/dense/
13. Chollet, F. (n.d.-b). *Keras Documentation: Earlystopping*. Keras. https://keras.io/api/callbacks/early_stopping/
14. *Sklearn.preprocessing.StandardScaler*. scikit. (n.d.-b). https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
15. TensorFlow, T. (2023, September 27). *Tf.keras.optimizers.legacy.optimizer : tensorflow V2.14.0*. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/legacy/Optimizer
16. scikit, scikit. (n.d.-a). *SKLEARN.METRICS. MEAN_SQUARED_ERROR*. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html
17. Chollet, F. (n.d.-c). *Keras documentation: Layer weight regularizers*. Keras. https://keras.io/api/layers/regularizers/
18. Dash. (n.d.). *Dash installation*. Plotly. https://dash.plotly.com/installation

# Appendix

```
# Import libraries
#-----General------#
import numpy as np
import pandas as pd
import os
import sys
import math
import random
import requests
import json

#-----Plotting-----#
import matplotlib.pyplot as plt
import plotly.graph_objects as go
import plotly.express as px
import plotly.offline as py
py.init_notebook_mode(connected=True)
import seaborn as sns
# from pandas_profiling import ProfileReport

#-----Utility-----#
import itertools
import warnings
warnings.filterwarnings("ignore")
import re
import gc
from bs4 import BeautifulSoup as soup
from urllib.request import Request, urlopen
from datetime import date, datetime,timedelta
!pip install fake-useragent
from fake_useragent import UserAgent
ua = UserAgent()

# Previous Data
# Function to convert string date to datetime object
def str_to_date(date_str):
    return datetime.strptime(date_str, '%Y-%m-%d')

# Function to convert datetime object to string date
def date_to_str(date_obj):
    return date_obj.strftime('%Y-%m-%d')
```

```
# Initialize parameters
base_url = 'https://www.ncdc.noaa.gov/cdo-web/api/v2/data'
api_key = 'csaPPKuENBgQQVGQTQFgMmUuDTGxdFpr'
headers = {'token': api_key,
           'User-Agent':ua.random,}

# Start and end dates for the entire data fetching period
start_date = str_to_date('2020-08-15')
end_date = str_to_date('2023-11-01')
2
# Adjust this depending on how much data you expect each day
days_per_request = 50

# DataFrame to hold all results
all_data_1 = pd.DataFrame()

while start_date < end_date:
    # Update end date for the request
    request_end_date = min(start_date + timedelta(days=days_per_request), end_date)

    # Set up parameters for the request
    params = {
        'datasetid': 'GHCND',
        'locationid': 'CITY:US360019',
        'stationid':'GHCND:USW00094728',
        'startdate': date_to_str(start_date),
        'enddate': date_to_str(request_end_date),
        'units': 'metric',
        'limit': 1000
    }

    # Make the request
    response = requests.get(base_url, headers=headers, params=params)

    # Check if the request was successful
    if response.status_code == 200:
        data = response.json()

        # Convert this part of the results to a DataFrame and append
        part_data = pd.json_normalize(data['results'])
        all_data_1 = pd.concat([all_data_1, part_data], ignore_index=True)

        # Update the start date for the next request
        start_date = request_end_date + timedelta(days=1)
```

```
    else:
        print(f"Error: {response.status_code} - {response.text}")
        break

# Make previous data as dataframe
df_prev = all_data_1.copy()

df_prev = df_prev.drop(columns=['attributes','station'])
df_prev = df_prev.pivot(index='date', columns='datatype', values='value')
df_prev = df_prev.drop(columns=['TSUN'])

# Check the consistency

# Define the start and end dates
start_date = datetime(2020, 8, 15)
end_date = datetime(2023, 11, 1)

# Calculate the number of days between the two dates
number_of_days = (end_date - start_date).days
print(number_of_days)

print(df_prev.shape)

# Get today's data
# Start and end dates for the entire data fetching period
start_date = str_to_date('2023-11-02')
end_date = datetime.now()
2
# Adjust this depending on how much data you expect each day
days_per_request = 50

# DataFrame to hold all results
all_data = pd.DataFrame()

while start_date < end_date:
    # Update end date for the request
    request_end_date = min(start_date + timedelta(days=days_per_request), end_date)

    # Set up parameters for the request
    params = {
        'datasetid': 'GHCND',
        'locationid': 'CITY:US360019',
        'stationid':'GHCND:USW00094728',
        'startdate': date_to_str(start_date),
```

```
        'enddate': date_to_str(request_end_date),
        'units': 'metric',
        'limit': 1000
    }

    # Make the request
    response = requests.get(base_url, headers=headers, params=params)

    # Check if the request was successful
    if response.status_code == 200:
        data = response.json()

        # Convert this part of the results to a DataFrame and append
        part_data = pd.json_normalize(data['results'])
        all_data = pd.concat([all_data, part_data], ignore_index=True)

        # Update the start date for the next request
        start_date = request_end_date + timedelta(days=1)
    else:
        print(f"Error: {response.status_code} - {response.text}")
        break

# Make today's data as df

df_today = all_data.copy()
df_today = df_today.drop(columns=['attributes','station'])
df_today = df_today.pivot(index='date', columns='datatype', values='value')
print(df_today.shape)
from datetime import datetime

# Define the start and end dates
start_date = datetime(2023, 11, 1)
end_date = datetime.now()

# Calculate the number of days between the two dates
number_of_days = (end_date - start_date).days
print(number_of_days)
# df_today = df_today.drop(columns=['PGTM'])

# Check the columms from two dataset

# Get the columns of each dataset
columns_df1 = set(df_prev.columns)
columns_df2 = set(df_today.columns)
```

```
# Compare columns
if columns_df1 == columns_df2:
    print("Both datasets have the same columns.")
else:
    print("Datasets do not have the same columns.")

    # To see which columns are different
    only_in_df1 = columns_df1 - columns_df2
    only_in_df2 = columns_df2 - columns_df1

    if only_in_df1:
        print("Columns only in the first dataset:", only_in_df1)
    if only_in_df2:
        print("Columns only in the second dataset:", only_in_df2)

# Try to combine them together
column_name = 'PGTM'  # Replace 'col' with your actual column name

# Check if the column exists in the DataFrame and drop it if it does
if column_name in df_today.columns:
    df_today = df_today.drop(columns=[column_name])
df_combine = pd.concat([df_prev,df_today],axis=0)
df_combine.index = pd.to_datetime(df_combine.index)

df_combine.shape

df_combine.info()

#  Handle multiple columns with missing data and fill them with the average values from the
corresponding day in previous years


average_values = df_combine.groupby([df_combine.index.month,
df_combine.index.day]).mean()

def impute_missing(row, avg_values):
    for col in df_combine.columns:
        if pd.isna(row[col]):
            month_day = (row.name.month, row.name.day)
            row[col] = avg_values.loc[month_day, col]
    return row
df_combine = df_combine.apply(impute_missing, axis=1, args=(average_values,))
```

```
df_combine.fillna(0, inplace=True)
df_combine.info()

# Feature engineering (using the regular method without)
df_combine['TAVG'] = (df_combine['TMAX'].astype(float)+df_combine['TMIN'].astype(float))/2
columns_to_shift = ['TAVG','TMAX', 'TMIN', 'PRCP', 'SNOW', 'AWND', 'RHAV', 'RHMN', 'RHMX',
'ADPT', 'ASLP', 'ASTP']

for col in columns_to_shift:
    for i in range(1, 4):  # Shift 1, 2, and 3 steps
        df_combine[f'{col}_{i}'] = df_combine[col].shift(i)


df_combine = df_combine.dropna()
scaled_df = df_combine
scaled_df.info()

# Model Training

import tensorflow as tf
from sklearn.model_selection import train_test_split,GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import EarlyStopping
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.optimizers.legacy import Adam
from sklearn.metrics import mean_absolute_error, mean_squared_error
from tensorflow.keras.regularizers import l2

df_train = scaled_df.drop(columns=['TAVG','TMIN','TMAX'], axis=1)
model_features = df_train
target = scaled_df['TAVG']

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(model_features, target, test_size=0.2,
random_state=42)

scaler_x = StandardScaler()
scale = scaler_x.fit_transform(X_train)
X_train = pd.DataFrame(scale, columns=X_train.columns,index=X_train.index)


scaler = StandardScaler()
scale = scaler.fit_transform(X_test)
```

```
X_test = pd.DataFrame(scale, columns=X_test.columns,index=X_test.index)
l2_lambda = 0.1

model = Sequential([
   Dense(64, activation='elu',
input_shape=(X_train.shape[1],),kernel_regularizer=l2(l2_lambda)),
   Dense(64, activation='elu',kernel_regularizer=l2(l2_lambda)),
   Dense(1)  # Output layer with one neuron and no activation for regression
])

early_stopping = EarlyStopping(monitor='val_loss', patience=10)

optimizer=Adam(learning_rate = 0.001)
model.compile(optimizer='adam', loss='mean_squared_error', metrics=['mse'])
model.fit(X_train, y_train, epochs=200, batch_size=32,
validation_split=0.2,callbacks=[early_stopping])


loss = model.evaluate(X_test, y_test)

print(f"Test Loss: {loss}")
predictions = model.predict(X_test)
print(f"Predictions: {predictions[:,0]}")
print(f"Actual: {y_test}")
mae = mean_absolute_error(y_test, predictions)
print(f"Mean Absolute Error: {mae}")

import matplotlib.pyplot as plt
# Plotting y_test vs y_pred_rf to compare the actual and predicted values
plt.figure(figsize=(10, 6))
plt.scatter(y_test.reset_index().index, y_test.values, label='Actual', color='blue', marker='o')
plt.scatter(y_test.reset_index().index, predictions, label='Predicted', color='red', marker='x')

plt.title('Comparison of Actual and Predicted Average Temp per Day (Testing set)')
plt.xlabel('Sample Index')
plt.ylabel('Temp')
plt.legend()
plt.grid(True)
plt.show()

# Plot the model

from keras.utils import plot_model
plot_model(model, to_file='model_plot.jpg', show_shapes=True, show_layer_names=True)
```

```
# Plot purpose
df_combine = df_combine.drop(df_combine.columns[25:],axis=1)
# Check for missing values
missing_values = df_combine.isnull().sum()

# Summary statistics for numerical features
summary_stats = df_combine.describe()

# Correlation matrix
corr_matrix = df_combine.corr()

# Plotting the distribution of the first few numerical features
plt.figure(figsize=(15, 5))
for i, column in enumerate(df_combine.columns[1:5], 1):  # Skip the date column
    plt.subplot(1, 4, i)
    sns.histplot(df_combine[column], kde=True)
    plt.title(f'Distribution of {column}')

plt.tight_layout()
plt.show()

missing_values, summary_stats.iloc[:, :5], corr_matrix.iloc[:5, :5]  # Displaying a subset for
clarity


# Plotting the correlation matrix heatmap
plt.figure(figsize=(15, 15))
sns.heatmap(corr_matrix, annot=False, cmap='coolwarm', linewidths=.5)
plt.title('Correlation Matrix Heatmap')
plt.show()


# Identifying highly correlated pairs with a threshold of 0.9
correlation_threshold = 0.9
highly_correlated_pairs = []

# Iterate over the correlation matrix
for i in range(len(corr_matrix.columns)):
    for j in range(i):
        if abs(corr_matrix.iloc[i, j]) > correlation_threshold:
            col_pair = (corr_matrix.columns[i], corr_matrix.columns[j], corr_matrix.iloc[i, j])
            highly_correlated_pairs.append(col_pair)
```

```
# Converting to a DataFrame for better readability
highly_correlated_pairs_df = pd.DataFrame(highly_correlated_pairs, columns=['Feature 1',
'Feature 2', 'Correlation'])
highly_correlated_pairs_df.sort_values(by='Correlation', ascending=False)


from pandas.tseries.offsets import DateOffset

# Get the last date in the dataset
last_date = scaled_df.index[-80]

# Calculate the date 30 days after today
today = pd.Timestamp.now().normalize()
thirty_days_after_today = today + DateOffset(days=30)

# Create a date range from the last date in the dataset to 30 days after today
extended_dates = pd.date_range(start=last_date + DateOffset(days=1),
end=thirty_days_after_today)

# Create a dataframe with these new dates and NaN values for the columns
extended_data = pd.DataFrame(index=extended_dates, columns=scaled_df.columns)

# # Remove the target feature
# extended_data = extended_data.drop(columns=['TAVG'])

# Display the last few rows to verify the extension
extended_data.info()


for date in extended_data.index:
    if date in scaled_df.index:
        extended_data.loc[date] = scaled_df.loc[date]
def impute_missing_pred(row, avg_values):
    for col in scaled_df.columns:
        if pd.isna(row[col]):
            month_day = (row.name.month, row.name.day)
            row[col] = avg_values.loc[month_day, col]
    return row
average_values_pred = scaled_df.groupby([scaled_df.index.month,
scaled_df.index.day]).mean()

extended_data = extended_data.apply(impute_missing_pred, axis=1,
args=(average_values_pred,))
```

```
scaled_df_pred = extended_data


scaled_df_pred.info()

scaled_df_pred = scaled_df_pred.drop(columns=['TAVG','TMIN','TMAX'], axis=1)
mock_data_normalized = scaler.transform(scaled_df_pred)
predictions_predictions = model.predict(mock_data_normalized)


predictions_predictions_flattened = [item for sublist in predictions_predictions for item in sublist]

fig_1 = go.Figure()

# Adding line for df1
fig_1.add_trace(go.Scatter(x=df_combine.index, y=df_combine['TAVG'], mode='lines',
name='Actual'))

# Adding line for df2
fig_1.add_trace(go.Scatter(x=scaled_df_pred.index, y=predictions_predictions_flattened,
mode='lines', name='Prediction'))

# Updating layout with range slider
fig_1.update_layout(
    xaxis_title='Date',
    yaxis_title='Value',
    xaxis=dict(
        rangeslider=dict(
            visible=True
        ),
        type='date'  # Assuming x-axis is datetime, change if otherwise
    )
)

# Show the plot
fig_1.show()


!pip install dash
import dash
from dash import dcc
from dash import html
from dash.dependencies import Input, Output
```

```
def generate_heatmap(selected_features):

    filtered_data = df_combine[selected_features]
    correlation_matrix = filtered_data.corr()
    fig = px.imshow(correlation_matrix,
                    labels=dict(x="Features_1", y="Features_2", color="Correlation"),
                    x=correlation_matrix.columns,
                    y=correlation_matrix.columns,
                    color_continuous_scale='pinkyl')
    fig.update_layout(title='Feature Correlation Heatmap', title_x=0.5)
    return fig




import base64

def encode_image(image_file):
    with open(image_file, 'rb') as file:
        return base64.b64encode(file.read()).decode('ascii')


import plotly.graph_objs as go

predictions_flattened = [item for sublist in predictions for item in sublist]

# Assuming y_test and predictions are your data
scatter_plot = go.Figure()

# Actual Values
scatter_plot.add_trace(go.Scatter(
    x=y_test.reset_index().index,
    y=y_test.values,
    mode='markers',
    name='Actual',
    marker=dict(color='blue')
))

# Predicted Values
scatter_plot.add_trace(go.Scatter(
    x=y_test.reset_index().index,
    y=predictions_flattened,
    mode='markers',
    name='Predicted',
```

```
    marker=dict(color='red', symbol='x')
))

# Updating layout
scatter_plot.update_layout(
    # title='Comparison of Actual and Predicted Average Temp per Day',title_x=0.5,
    xaxis_title='Sample Index',
    yaxis_title='Temp',
    legend_title='Legend',
    template='plotly_white'
)


from dash import Dash, html, dcc

app = Dash(__name__)

app.layout = html.Div([
    html.H1('DATA 602 Final Project Weather Forcasting in Central Park', style={'textAlign':
'center'}),

    # Dropdown and line chart for time series
    dcc.Dropdown(
        id='column-dropdown',
        options=[{'label': col, 'value': col} for col in df_combine.columns],
        value='TAVG'
    ),
    dcc.Graph(id='line-chart'),

    # Flex container for heatmap and model diagram
    html.Div([
        # Heatmap
        html.Div([
            html.H2('Select Features for Correlation Heatmap'),
            dcc.Dropdown(
                id='feature-select-dropdown',
                options=[{'label': col, 'value': col} for col in df_combine.columns],
                value=df_combine.columns.tolist()[:5],  # Default selected values
                multi=True
            ),
            dcc.Graph(id='feature-correlation-heatmap')
        ], style={'flex': '1'}),

        # Model Diagram
```

```
        html.Div([
            html.H2('Simple model diagram'),
            html.Img(src='{}'.format(encode_image('model_plot.jpg')),
                     style={})
        ], style={'flex': '1'})
    ], style={'display': 'flex', 'flex-direction': 'row', 'align-items': 'stretch', 'textAlign': 'center', 'flex-
flow': 'space-evenly', 'margin-bottom': '20px'}),  # Added margin-bottom here

    # predicted map
    html.Div([
        html.H2('Comparison of Actual and Predicted Average Temp per Day',style={'textAlign':
'center'}),
        dcc.Graph(figure=scatter_plot)
    ], style={'flex': '1', 'flex-flow': 'space-evenly'}),
    html.Div([
        html.H2('Weather forcasting for Next few days',style={'textAlign': 'center'}),
        dcc.Graph(figure=fig_1)
    ], style={'flex': '1', 'flex-flow': 'space-evenly'})

], style={'flex': '1', 'flex-flow': 'space-evenly'})


@app.callback(
    Output('line-chart', 'figure'),
    [Input('column-dropdown', 'value')]
)
def update_chart(selected_column):
    fig = px.line(df_combine, x=df_combine.index, y=['TAVG', selected_column])
    fig.update_layout(title='Comparison of TAVG and Features Column', title_x=0.5)

    return fig
# callback for heatmap
@app.callback(
    Output('feature-correlation-heatmap', 'figure'),
    [Input('feature-select-dropdown', 'value')]
)
def update_heatmap(selected_features):
    return generate_heatmap(selected_features)


if __name__ == '__main__':
    app.run_server(debug=True,host="127.0.0.1", port=8050)
```