STATS604 Project 4: Weather Prediction

Kevin Jin, Kevin Liang, Noah Kochanski, Jiuqian Shang, Ashlan Simpson

12/8/24

1 Introduction

Weather forecasts are an important, yet often overlooked, luxury of the modern world. We can easily get relatively accurate weather forecasts for up to two weeks into the future. Traditionally, weather forecasts are built off of physics-based models which requires a large computation overhead. In this project, we explore the use of applying statistical methods instead of the traditional models applied. Our goal is to, as accurately as possible, provide 5 day forecasts for 20 airports across the United States.

While it may be intuitive to leverage the plethora of historical weather data for all 20 locations, this probably isn't feasible given the time and computation for scraping data from the internet. Additionally, there are some self-imposed restrictions in terms of memory and computational costs associated with our prediction models. Through our project, by leveraging a relatively short period of historical data, we determine if statistical methods accurately and reliably forecast weather.

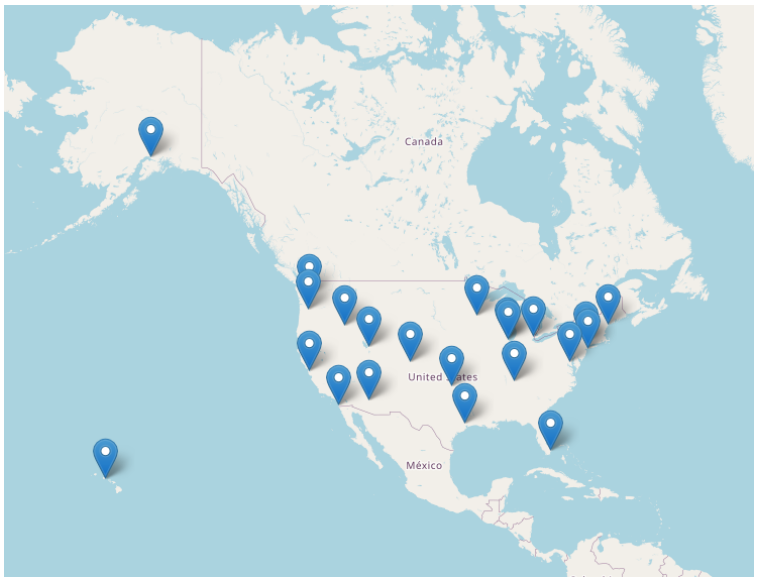

Figure 1: Map of all 20 airport locations to be forecasted.

2 Historical Data Collection

Weather is one of the most widely and longest collected forms of data. As such, there are plenty of reliable data sources which could aid in our project. We opted to scrape Weather Underground data using the package RSelenium. We collected historical weather data for our selected 20 locations across the United States, focusing on variables such as temperature, precipitation, dew point, wind speed, visibility and sea level pressure. The extraction of historical weather data from the Weather Underground website was conducted

using RSelenium, a browser automation tool that facilitated efficient scraping of the required information. Weather Underground was chosen as the data source because it provides ground truth data by the definition of our project, despite its use of Java-based elements, which make it more challenging to scrape. The code successfully collected information on location, date, high, low, and day-average temperatures, precipitation, dew point, wind speed, visibility, and sea level pressure. Spanning multiple locations and dates, the dataset offers a robust foundation for analyzing weather trends.

During the data collection process, a significant anomaly was identified on November 8, 2020, when no data was scrapable for any location with the current code. While the specific cause remains unclear and further investigation could clarify the issue, this was not pursued further. Notably, this date coincided with the announcement of the results of the 2020 U.S. presidential election, though the relevance of this timing is uncertain. Additionally, an examination of the data revealed occasional issues with missing values. On the Weather Underground platform, unrecorded hourly data was represented as zeros. For summary temperature metrics, this resulted in incorrect entries where high or low temperatures were recorded as zero, unless zero fell within the plausible range for the given day. Similarly, day-average temperatures appeared to incorporate zeros as placeholders for missing hourly data, leading to skewed averages.

These errors were identified through code errors and spot checking, but the code was not adjusted. Dates that could not be scrapped were skipped rather than attempted with a second set of scrapping code nested inside the first. Additionally, missing hourly data was not accounted for in the code as it would have taken the code identifying the issue, pulling information from the hourly table, correcting the affected values, and replacing them. While it was possible to add functionality to the code to address these issues, the short term nature of the project in conjunction with the low frequency of occurrence and ease of manual adjustments lead to the decision to apply corrections after the fact.

While the code worked very well, there was an issue applying it. The Chicago airport that we attempted to scrape did not have historical data leading to a scrapping error where the same data was pulled continuously. This led to a large error for Chicago. Similarly, the Denver airport of interest was not available for historical data. There were difficulties switching between the alternative city, BLANK City, for historical data to the proper Denver airport for updating data. These are issues that could have been minimized or corrected if identified earlier and is an area where we have room for improvement.

3 Exploratory Data Analysis

After compiling the data, we conducted an initial exploratory analysis to identify trends, patterns, and any potential anomalies that could influence our predictions.

The temperature trends and distributions vary significantly across cities, as observed in the daily temperature trend plot, Figure 2, and the boxplot, Figure 3. For example, some cities, like San Francisco (KSFO) and Miami (KMIA), experience higher and more consistent temperatures, while others, like Minneapolis (KMSP), New York (KJFK) and Alaska (PANC) show much lower temperatures with relatively high variabilities. This suggests that city-specific models would be more effective than a generalized approach to capture regional weather patterns accurately.

In in the daily temperature trend plot (Figure 2) and the boxplot (Figure 3), we can also see Chicago's data shows constant daily values, indicating potential data quality issues.

The precipitation barplot, Figure 4, highlights substantial differences in average precipitation across cities. For example, cities like Seattle (KSEA) has much higher average precipitation, consistent with its known rainy climates, while cities like Phoenix (KPHX) and San Diego (KSAN) exhibit minimal precipitation. These variations emphasize the need to model precipitation separately for each region.

The correlation heatmap, Figure 5, highlights non-linear relationships between key weather variables, such as the strong correlation between dew point and temperature metrics but weaker or non-linear associations with precipitation and wind speed. This complexity supports the use of non-linear models for better predictive performance.

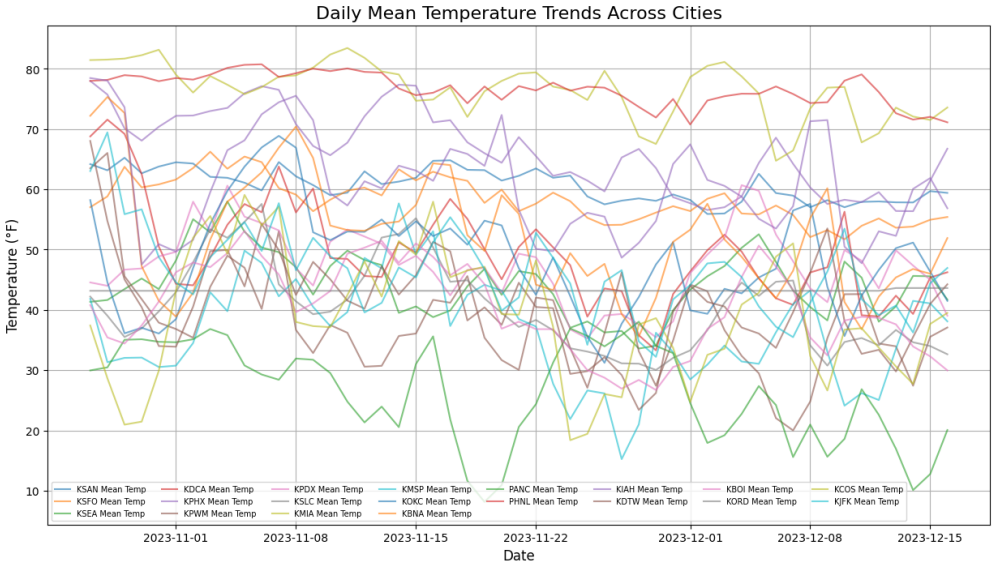

Figure 2: my caption

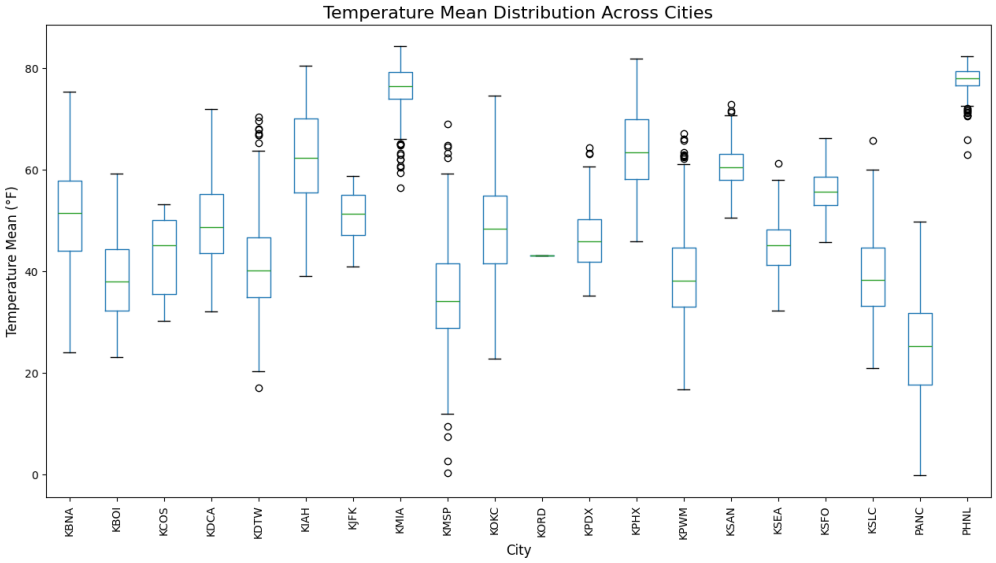

Figure 3: Boxplot of the mean temperature across the different cities.

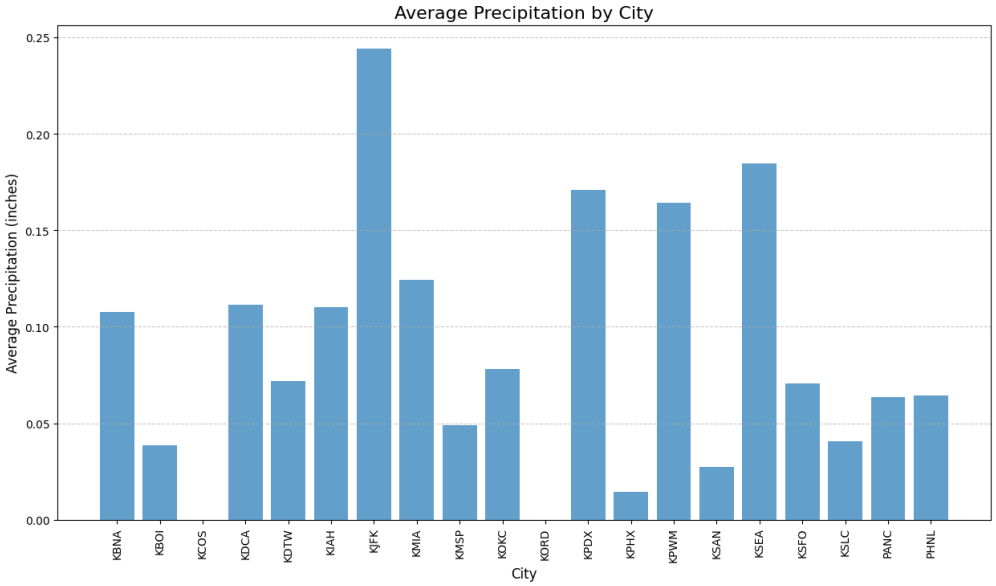

Figure 4: Average precipitation by city. High variability in precipitation with some citis with a lot of precipitation and other cities having no precipitation at all.

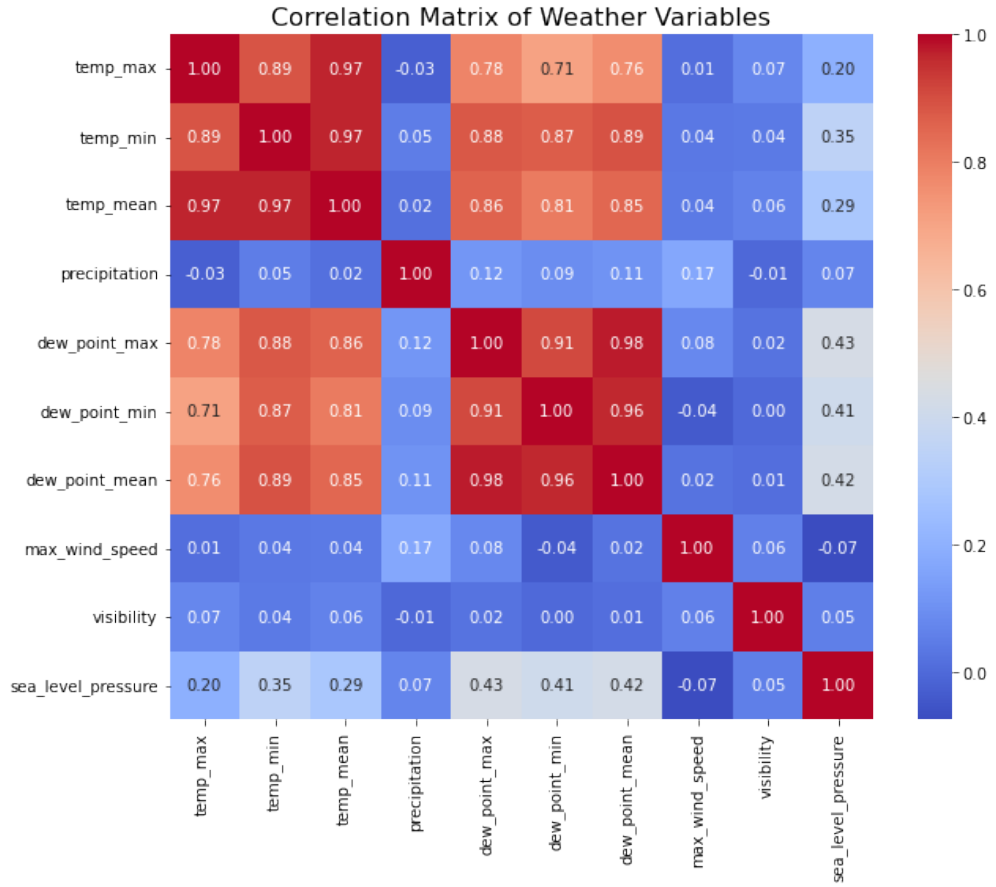

Figure 5: Correlation matrix of all weather covariates available.

These findings guide the modeling approach, emphasizing the need for city-specific, non-linear models and careful handling of data inconsistencies.

4 Methods

4.1 Modeling Approach

As determined from the EDA section, we wanted to have a city specific model that was nonlinear due to the complex relationships in the data. Thus, we decided to use a Gaussian Process Regression (GPR) model. GPR does not assume a functional form (e.g. linear) form of the data, as the data is modeled as distributions of functions, which we found was perfect for our case. As we also were dealing with time series data, we also wanted a multi-output response to predict all of the predictors for a future date in order to extrapolate our predictions beyond just one day ahead predictions. GPR provides a nonparametric method for this task, and ideally, also comes with its own uncertainty quantification of its results. While this was something we did not take advantage of in our model, in future work, this is a direction we can focus on re-weight our predictions if we are not confident in it.

When using GPR, we are able to choose many different kinds of kernels. We ended up using a Radial Basis Function (RBF) kernel because while it is extremely smooth, this smoothness is able to capture the local dependence structure of our data that we will tune via cross validation as we will see. Thus, we had the following pipeline in mind.

1. First, we will train on the historical weather data from October to December that we had.
2. We will do hyperparameter tuning of the α parameter in GPR and and Length scale parameter in the RBF.
3. We will use RMSE as our loss for training and we chose to use a Lag of 5 days (with some motivation of this number chose in the prediction section).

4.2 Cross-Validation Training

To do the task in point 2, we performed cross-validation training by using the `TimeSeriesSplit` from `scikit-learn`. In this `TimeSeriesSplit`, we chose to use 5 splits (6 total folds). Then for the iteration $k \in [5]$, we will use the first k folds as our training set, and the remaining folds as our testing set. This means that for each successive iteration, our training set will be a super set of the previous iteration's training set. As alluded to before, the specific hyperparameters that we wanted to tune was the α parameter in GPR and ℓ which is the Length scale parameter in the RBF.

For some high level understanding of what these parameters control, α used in GPR can be thought of as the variance of the Gaussian measurement noise on the training observations. We were tuning α on the log scale, from 10^{-12} to 10^{10}. As we are treating α as a scalar, we are assuming the data comes an isotropic multivariate Gaussian i.e. that the variance is homeostatic for a city, which is a reasonable assumption given our initial data exploration.

On the other hand, ℓ controls how the local versus global structure of our kernel is going to be. In other words, it determines how far the influence of a single data extends to. Smaller ℓ captures the more rapid sharp changes in the data, but larger ℓ tends to smooth out much of the peaks and fluctuation in the data. We also tuned ℓ in the log space to range from 10^{-5} to 10^{5}. Thus, we had a final grid of hyper parameters in which we chose the pair that had the lowest validation RMSE. For all of the 20 models, we show an example of the cross validation training results below in Figure 6.

Only 100 configurations of the hyperparameters are shown here, and we see that the all of the models generally minimize the RMSE at around the same hyperparameters (besides Chicago/KORD at the bottom). This pattern generally occurred at around every 80 iterations, though it was never minimized more than the model hyperparameters shown in Figure 6.

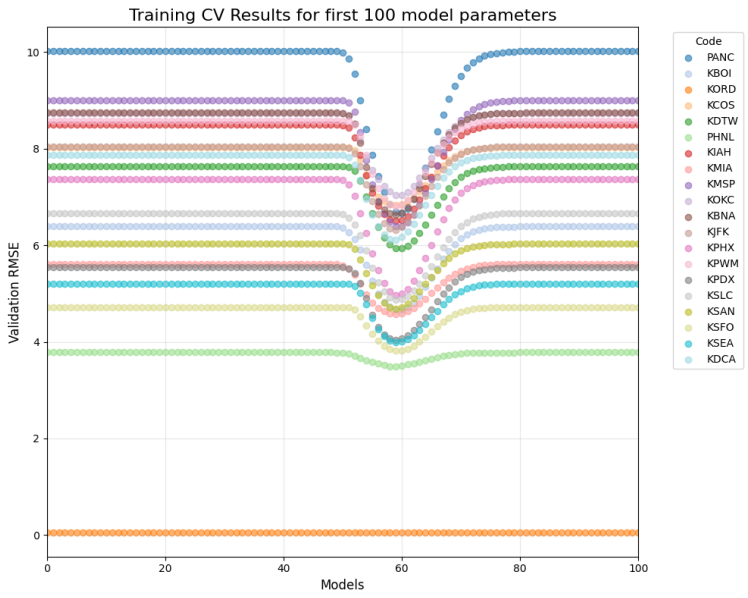

Figure 6: Validation RMSE of the first 100 models tested for each city

4.3 Framework for Predictions

After training our models, we now needed a framework to make predictions for the next 5 days, as our model only outputs a prediction for a given day. Thus, we followed this framework:

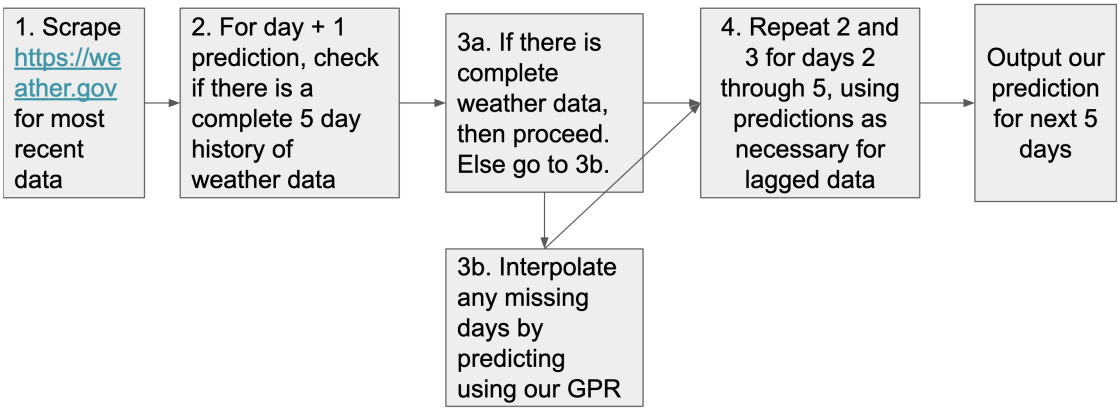

Figure 7: Flowchart outlining the entire analysis pipeline for 5 day out prediction.

Notice that in the first step of this framework, we will only be scraping current data from weather.gov. On this site, only the most recent 3 days of weather history is given. As such, this motivated our use of using the 5 past days of history to predict the future, as if are predicting further out into the future, we will need to interpolate the days prior to 3 days before our desired data, so we wanted to maximize our use of current data while still trying to incorporate lagged data.

5 Results

Using our predictions from November 26th, 2024 to December 5h, 2024, the RMSE of all of predictions combined for each day was around 10.6. If remove Chicago from our results, our RMSE decreased to around 8.93. To see how well we did on each city, we plotted our results using on a per city basis in Figure 8. As we

can see from this figure, we performed the worst when predicting Chicago's (KORD) weather. After Chicago, the next highest RMSEs were New York (KJFK), Minneapolis (KMSP), and Anchorage (KPANC). This makes sense as the winter storm that occurred during Thanksgiving should have affected our predictions.

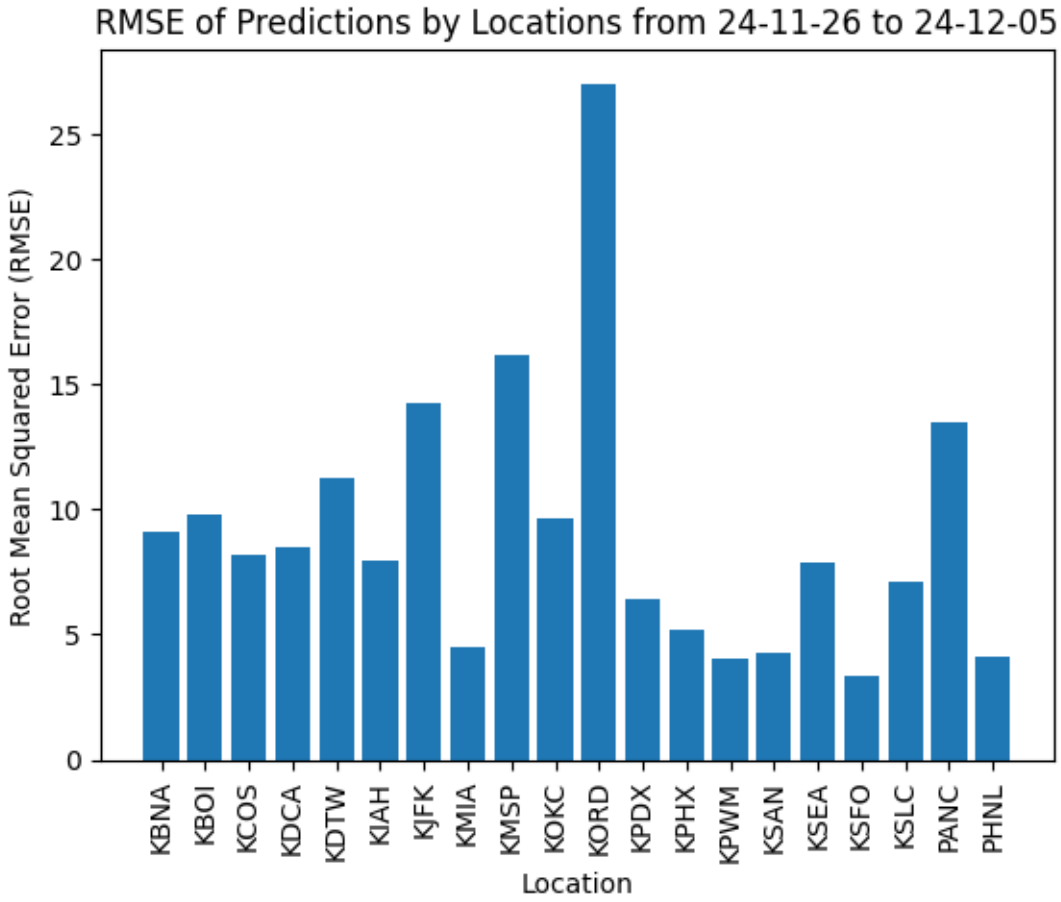

Figure 8: RMSE of Predictions from November 26th to December 5th.

On the other hand, our model performed quite well with cities like Miami (KMIA), Portland (KPWM), and San Francisco (KSFO). This coincides with pre-existing knowledge that these cities are quite stable in climate and weather (at least temperature wise) as these are quite attractive places to go during the winter.

Beyond just looking at predictions at a city level, we also compared our predictions with the naive method of just using the previous day to predict 1 through 5 days out in Figure 9.

We can see from this figure that besides 1 Days out, our model had a lower RMSE in our predictions than using the naive method. This suggests that for longer term predictions, a naive method of using the most recent day to predict is not as beneficial as our model. However, we do note that the sample variance of SSE over all models suggests that there potentially is not much of a distinction between the two methods.

6 Conclusion

Looking at the prediction results, we had significant issues with the cities Chicago and Denver. The reason for this is due to mismatch errors between airports. For Denver, KDEN does not have any historical data recorded on Weather Underground so as an alternative we pulled in historical data from a nearby airport in Colorado Springs. However, when we are pulling in current data, on weather.gov we can successfully pull

	RMSE of Predictions	
Days Out	**GPR**	**Most Recent Day**
1	9.81	7.30
2	9.97	10.33
3	11.78	12.59
4	11.23	13.85
5	11.66	15.21

Figure 9: RMSE of Predictions based on how far out they were.

data for KDEN but there was a mismatch between airport names so we predicted the same temperature for every single day past 11/25. Similarly for Chicago, the historical data was KORD but the current data is pulling from a different smaller airport in Chicago. Once these errors are fixed, there will be an improvement for our prediction results.

For future improvements of our project, perhaps we were too optimistic with using only historical data for the months of October to December. It's very likely that we did not capture any long term weather patterns that could help us with predictions. In addition, the implementation of the Gaussian process regression in sklearn treats multi-output responses as independent of one another and perhaps looking at an alternative implementation may improve our results. One major thing that we would reconsider is to possibly assimilate more observational data from sources that are more easily accessible through simple Python packages. Although working with data from Weather Underground has certain advantages, it has its own difficulties in scraping. Other weather data sources such as open-meteo have data that is much more easily accessible. Additionally, having a 0.1 to 1 degree difference compared to Weather Underground will most likely not have a significant impact on the prediction results. Lastly, our data never updates past November 25th and we used an interpolation method to fill in missing dates, which may not the best approach. Ideally, we would store the current data that is being pulled each day instead of doing an interpolation technique.

7 Code Appendix

7.1 EDA

```
# %%
import pandas as pd
import numpy as np
import os, pickle, janitor, requests
from warnings import simplefilter
simplefilter(action="ignore", category=pd.errors.PerformanceWarning)

# %%
## Get Location Code from the Data Frame
def getLocationCode(text):
    return text.split("/")[-1]

## Apply it to a Pandas Series
def getLocationCodeSeries(pd_series):
    return pd_series.apply(getLocationCode)

## Get the Data Frame
def getDataFrame(url: str):
    df = (
        pd.read_csv(url)
        .pipe(janitor.clean_names, strip_underscores=True)
        .assign(date=lambda x: pd.to_datetime(x.date),
                location=lambda x: getLocationCodeSeries(x.location))
        .drop(columns=["unnamed_0", "actual_time", "dew_point"])
        .rename(columns={"high_temp": "temp_max",
                         "low_temp": "temp_min",
                         "day_average_temp": "temp_mean",
                         "high": "dew_point_max",
                         "low": "dew_point_min",
                         "average": "dew_point_mean",
                         "visibiilty": "visibility",})
    )
    return df

# %%
raw_data_url = "https://raw.githubusercontent.com/kevinmjin/winter_weather_data/refs/heads/main/winter_

ogdf = getDataFrame(f'{raw_data_url}')
mydf = ogdf.copy()

# %% [markdown]
# ## Datacleaning

# %%
mydf.describe()

# %%
mydf[mydf['temp_max'] == 201]

# %%
```

```
mydf.loc[3317, 'temp_max'] = 50

# %%
mydf[mydf['temp_min'] == 0]

# %%
missing_min = mydf[mydf['temp_min'] == 0]  # Example subset, you can define your own subset
missing_min_location = missing_min['location'].tolist()
missing_min_date = missing_min['date'].tolist()
missing_min_rows = missing_min.index.tolist()

# %%
min_refills = [36, # KSFO 2020-12-01
               43, # KSEA 2022-11-25
               38, # KDCA 2021-11-09
               45, # KPWM 2019-10-27
               73, # KMIA 2020-11-13
               72, # KMIA 2023-11-18
               73, # KMIA 2023-11-25
               75, # KMIA 2024-11-11
               36, # KSWF 2019-11-27
               39, # KSWF 2019-12-14
               30, # KSWF 2020-11-17
               41, # KSWF 2020-11-21
               46, # KSWF 2020-11-27
               37, # KSWF 2020-11-28
               30, # KSWF 2020-12-03
               43, # KSWF 2021-12-16
               39, # KSWF 2022-11-03
               32, # KSWF 2024-11-16
               0, # PANC 2020-12-05
               0, # PANC 2021-11-16
               0,# PANC 2022-11-28
               0, # PANC 2022-11-07
               33, # KBOI 2019-11-13
               32, # KBOI 2021-12-02
               31, # KJFK 2019-12-02
               52, # KJFK 2020-10-28
               ]

# %%
# Refill unreasonable values
for i in range(len(missing_min_rows)):
    mydf.loc[missing_min_rows[i], 'temp_min'] = min_refills[i]
    print(missing_min_location[i], missing_min_date[i],'refilled with temp_min= ', min_refills[i])


# %%
locations_dict = {
                "Anchorage": "PANC",
                "Boise": "KBOI",
                "Chicago": "KORD",
                "Denver": "KCOS",
```

```
                "Detroit": "KDTW",
                "Honolulu": "PHNL",
                "Houston": "KIAH",
                "Miami": "KMIA",
                "Minneapolis": "KMSP",
                "Oklahoma City": "KOKC",
                "Nashville": "KBNA",
                "New York": "KJFK",
                "Phoenix": "KPHX",
                "Portland ME": "KPWM",
                "Portland OR": "KPDX",
                "Salt Lake City": "KSLC",
                "San Diego": "KSAN",
                "San Francisco": "KSFO",
                "Seattle": "KSEA",
                "Washington DC": "KDCA"
            }
## keep only the locations of locations_dict_values
mydf = mydf[mydf['location'].isin(locations_dict.values())]
mydf.to_csv('data/data_cleaned.csv')

# %% [markdown]
# ## EDA

# %% [markdown]
# ### Time Series Analysis

# %%
import matplotlib.pyplot as plt
mydf['date'] = pd.to_datetime(mydf['date'])
mydf['year']= mydf['date'].dt.year
cities = mydf['location'].unique()

plt.figure(figsize=(15, 8))
for city in cities:
    city_data = mydf[mydf['location'] == city][mydf['year'] == 2023]
    plt.plot(city_data['date'], city_data['temp_mean'], label=f'{city} Mean Temp', alpha=0.6)

plt.title("Daily Mean Temperature Trends Across Cities", fontsize=16)
plt.xlabel("Date", fontsize=12)
plt.ylabel("Temperature (°F)", fontsize=12)
plt.legend(loc='upper right', fontsize=10)
plt.grid()
plt.show()


# %% [markdown]
# ### Correlations

# %%
import seaborn as sns

numerical_columns = ['temp_max', 'temp_min', 'temp_mean', 'precipitation',
```

```
                    'dew_point_max', 'dew_point_min', 'dew_point_mean',
                    'max_wind_speed', 'visibility', 'sea_level_pressure']
correlation_data = mydf[numerical_columns]

# Calculate the correlation matrix
correlation_matrix = correlation_data.corr()

# Plot the heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap="coolwarm", cbar=True)
plt.title("Correlation Matrix of Weather Variables", fontsize=16)
plt.show()

# Scatter plots for key correlations (Example)
# Plot temp_mean vs. dew_point_mean
plt.figure(figsize=(8, 6))
plt.scatter(mydf['dew_point_mean'], mydf['temp_mean'], alpha=0.6)
plt.title("Scatter Plot: Temp Mean vs. Dew Point Mean", fontsize=16)
plt.xlabel("Dew Point Mean", fontsize=12)
plt.ylabel("Temperature Mean (°F)", fontsize=12)
plt.grid()
plt.show()

# Plot temp_mean vs. sea_level_pressure
plt.figure(figsize=(8, 6))
plt.scatter(mydf['sea_level_pressure'], mydf['temp_mean'], alpha=0.6)
plt.title("Scatter Plot: Temp Mean vs. Sea Level Pressure", fontsize=16)
plt.xlabel("Sea Level Pressure", fontsize=12)
plt.ylabel("Temperature Mean (°F)", fontsize=12)
plt.grid()
plt.show()


# %% [markdown]
# ### City-wise Analysis
#

# %%

city_summary = mydf.groupby('location').agg({
    'temp_min': [ 'median', 'std'],
    'temp_mean': [ 'median', 'std'],
    'temp_max': ['median', 'std'],
    'precipitation': 'mean',
    'max_wind_speed': 'mean'
}).reset_index()


city_summary.columns = ['_'.join(col).strip('_') for col in city_summary.columns]

print("City-wise Weather Summary:")
print(city_summary)
```

```
# Boxplot for temperature distributions across cities
plt.figure(figsize=(15, 8))
mydf.boxplot(column='temp_mean', by='location', grid=False, rot=90, figsize=(15, 8))
plt.title("Temperature Mean Distribution Across Cities", fontsize=16)
plt.xlabel("City", fontsize=12)
plt.ylabel("Temperature Mean (°F)", fontsize=12)
plt.suptitle("")  # Remove automatic title
plt.show()

# Bar chart: Average precipitation across cities
city_avg_precipitation = mydf.groupby('location')['precipitation'].mean().reset_index()

plt.figure(figsize=(15, 8))
plt.bar(city_avg_precipitation['location'], city_avg_precipitation['precipitation'], alpha=0.7)
plt.title("Average Precipitation by City", fontsize=16)
plt.xlabel("City", fontsize=12)
plt.ylabel("Average Precipitation (inches)", fontsize=12)
plt.xticks(rotation=90)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()




# %%
import folium
import pandas as pd

airport_data = {
    'ICAO': ['KSAN', 'KSFO', 'KSEA', 'KDCA', 'KPHX', 'KPWM', 'KPDX', 'KSLC',
             'KMIA', 'KMSP', 'KOKC', 'KBNA', 'KSWF', 'PANC', 'KMDW', 'PHNL',
             'KIAH', 'KDTW', 'KBOI', 'KORD', 'KCOS', 'KJFK'],
    'Name': ['San Diego Intl', 'San Francisco Intl', 'Seattle-Tacoma Intl',
             'Reagan National', 'Phoenix Sky Harbor', 'Portland Intl Jetport',
             'Portland Intl', 'Salt Lake City Intl', 'Miami Intl', 'Minneapolis-St. Paul Intl',
             'Will Rogers World', 'Nashville Intl', 'Stewart Intl', 'Ted Stevens Anchorage Intl',
             'Chicago Midway', 'Honolulu Intl', 'George Bush Intercontinental',
             'Detroit Metropolitan', 'Boise Airport', 'Chicago O\'Hare',
             'Colorado Springs', 'John F. Kennedy Intl'],
    'Latitude': [32.7338, 37.6188, 47.4502, 38.8512, 33.4353, 43.646, 45.5898,
                 40.7861, 25.7933, 44.8848, 35.3931, 36.1263, 41.5041, 61.1743,
                 41.785, 21.3187, 29.9902, 42.2162, 43.5644, 41.9742, 38.8058, 40.6413],
    'Longitude': [-117.1933, -122.375, -122.3088, -77.0377, -112.0078, -70.3081,
                  -122.5951, -111.9776, -80.2906, -93.2223, -97.6016, -86.6774,
                  -74.1048, -149.998, -87.7524, -157.9224, -95.3368, -83.3554,
                  -116.223, -87.9073, -104.7007, -73.7781]
}

# Convert to DataFrame
df = pd.DataFrame(airport_data)

# Base map
usa_map = folium.Map(location=[39.8283, -98.5795], zoom_start=4)
```

```
# Add airport markers
for _, row in df.iterrows():
    folium.Marker(
        location=[row['Latitude'], row['Longitude']],
        popup=f"{row['ICAO']} - {row['Name']}",
        tooltip=f"{row['ICAO']}",
    ).add_to(usa_map)

usa_map.save("airport_map.html")
```

7.2 Model Training

```
# %%
import pandas as pd
import numpy as np
import os, pickle, janitor

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.gaussian_process import GaussianProcessRegressor as GPR
from sklearn.gaussian_process.kernels import RBF

from warnings import simplefilter
simplefilter(action="ignore", category=pd.errors.PerformanceWarning)


## For the scraping
import requests
from bs4 import BeautifulSoup
from datetime import datetime, timedelta



# %%
def getCleanDataFrame(folder_path: str):
    files = os.listdir(folder_path)
    ## keep only the csv files
    files = [file for file in files if file.endswith(".csv")]
    df = pd.concat([pd.read_csv(folder_path + file)
                    .pipe(janitor.clean_names, strip_underscores=True)
                    .drop(columns=["unnamed_0"])
                    .assign(date=lambda x: pd.to_datetime(x.date),
                            temp_max=lambda x:pd.to_numeric(x.temp_max),
                            temp_min=lambda x:pd.to_numeric(x.temp_min),
                            temp_mean=lambda x:pd.to_numeric(x.temp_mean),
                            precipitation=lambda x:pd.to_numeric(x.precipitation),
                            dew_point_max=lambda x:pd.to_numeric(x.dew_point_max),
                            dew_point_min=lambda x:pd.to_numeric(x.dew_point_min),
                            dew_point_mean=lambda x:pd.to_numeric(x.dew_point_mean),
                            max_wind_speed=lambda x:pd.to_numeric(x.max_wind_speed),
                            visibility=lambda x:pd.to_numeric(x.visibility),
                            sea_level_pressure=lambda x:pd.to_numeric(x.sea_level_pressure)
```

```
                    )
                     for file in files])
    return df

# %%
## Created Lagged Data
def allLaggedDataAvail(df: pd.DataFrame, lag: int, to_lag_columns: list):
    df = df.sort_values(by=["date"])
    df['day_diff'] = df['date'].diff().dt.days
    df['has_all_lagged'] = (
        df['day_diff']
        .rolling(window=lag)
        .apply(lambda x: np.all(x == 1), raw=True)
    )

    for col in to_lag_columns:
        for i_lag in range(1, lag + 1):
            df[f'{col}_lag_{i_lag}'] = df[f'{col}'].shift(i_lag)
    df = df[df['has_all_lagged'] == 1].copy()
    df.dropna(subset=[f'{col}_lag_{lag}' for lag in range(1, lag + 1)], inplace=True)

    # Drop temporary columns
    df = df.drop(columns=['day_diff', 'has_all_lagged'])
    #df = df.drop(columns=to_lag_columns[3:])
    return df.sort_values(by = "date").reset_index(drop=True)

## Create a Model for each Location
class LocationModel():
    def __init__(self, location: str, df: pd.DataFrame, lag: int):
        self.to_lag_columns = ['temp_max',
                               'temp_min',
                               'temp_mean',
                               'precipitation',
                               'dew_point_max',
                               'dew_point_min',
                               'dew_point_mean',
                               'max_wind_speed',
                               'visibility',
                               'sea_level_pressure']
        self.lag = lag

        self.location = location
        self.df = df.copy()
        self.lagged_df = allLaggedDataAvail(df, lag, self.to_lag_columns)

    def fit(self):
        X_ = (
            self.lagged_df
            .drop(columns=self.to_lag_columns)
            .drop(columns = ["date", "location"])
            .to_numpy()
        )
        y_ = self.lagged_df[self.to_lag_columns].to_numpy()
```

```
        self.scaler = StandardScaler()
        X_ = self.scaler.fit_transform(X_)

        ## Define the Kernel
        gp = GPR(normalize_y=True, n_restarts_optimizer=10)

        ## Define the Parameters
        param_grid = {"alpha": np.logspace(1e-12,10,100),
                      "kernel": [RBF(l, length_scale_bounds="fixed") for l in np.logspace(-5,5,100)]}
        ## Define the Grid Search
        tscv = TimeSeriesSplit(n_splits=5)
        self.model = GridSearchCV(gp, param_grid=param_grid, cv=tscv, n_jobs=-1,
                                  scoring = "neg_mean_squared_error").fit(X_, y_)
        return self


    def predict(self, date, scraped_data = None):
        ## If we are including the recent scraped data, append it into our data frame
        ## for better predictions
        if scraped_data is not None:
            bool_vec = np.zeros(len(scraped_data), dtype = bool)
            for i in range(len(scraped_data)):
                scraped_date = scraped_data.iloc[i]["date"]
                if scraped_date not in self.df["date"].values:
                    bool_vec[i] = True
            self.df = pd.concat([self.df, scraped_data[bool_vec]])
            self.df = self.df.sort_values(by = "date")

        ## check if date is string
        if isinstance(date, str):
            date = pd.to_datetime(date)
        ## check if date is datetime
        if not isinstance(date, pd.Timestamp):
            raise ValueError("Date must be a string or a pd.Timestamp object")

        ## Clearly, if date is already present in dataframe, we can just return the stored values
        if date in self.df["date"].values:
            return self.df[self.df.date == date][self.to_lag_columns].to_numpy()
        else:
            ## check if last lagged days are in the dataframe
            prior_date_range = pd.date_range(end = date - pd.Timedelta(days=1), periods = self.lag + 1,
            for day in prior_date_range:

                if day not in self.df["date"].values:
                    _ = self.predict(date=day)


            ## Now, we can predict the values for the date after previous lagged day values are present
            new_X_ = (
                self.df[(self.df.date >= prior_date_range[0]) & (self.df.date <= prior_date_range[-1])]
                .copy()
            )
```

```
            lagged_new_X = (
                allLaggedDataAvail(df = new_X_, lag = self.lag, to_lag_columns = self.to_lag_columns)
                .drop(columns = self.to_lag_columns)
                .drop(columns = ["date", "location"])
            )

            try:
                predicted_y = self.model.predict(self.scaler.transform(lagged_new_X.to_numpy()))[-1]
            except:
                ## fill missing values in lagged_new_X with 0
                lagged_new_X = lagged_new_X.fillna(0)
                predicted_y = self.model.predict(self.scaler.transform(lagged_new_X.to_numpy()))[-1]

            ## clip precipitation to be min 0
            if predicted_y[3] < 0:
                predicted_y[3] = 0

            ## clip visibility to be max 10
            if predicted_y[8] > 10:
                predicted_y[8] = 10

            ##
            predicted_obs = pd.DataFrame(predicted_y.reshape(1,10), columns = self.to_lag_columns)
            predicted_obs["date"] = date
            predicted_obs["location"] = self.location
            self.df = pd.concat([self.df, predicted_obs])
            self.df = self.df.sort_values(by = "date").reset_index(drop=True)
            return predicted_y

    def __str__(self):
        return f"{self.location} Model"


class WeatherForecast():
    def __init__(self, folder_path: str, lag: int):
        self.df = getCleanDataFrame(folder_path)
        self.locations = self.df.location.unique()
        self.models = {location: LocationModel(location, self.df[self.df.location == location], lag).fi

    def getCurrenHourlyWeatherData(self, place: str):
        col_names = ['date', 'Time', 'wind', 'visibility', 'Weather', 'Sky Condition', 'temp', 'dew_poi
                     'Min Temp 6hr', 'Rel Humidity', 'Wind Chill', 'Heat Index', 'sea_level_pressure', '
                     'precip_1hr', 'Precip 3hr', 'Precip 6hr']
        url = f"https://forecast.weather.gov/data/obhistory/{place}.html"

        response = requests.get(url)
        soup = BeautifulSoup(response.text, 'html.parser')
        table = soup.find('table', class_ = 'obs-history')
        for weather_data in table.find_all('tbody'):
            rows = weather_data .find_all('tr')
            data_dict = {col: [] for col in col_names}

            for row in rows:
```

```
                for col in col_names:
                    data_dict[col].append(row.find_all('td')[col_names.index(col)].text)
            val_list = []
            for i in data_dict["wind"]:
                val = i.split("\n")[1].replace(" ", "").replace("\n", "")
                if val == "":
                    val = "0"
                val_list.append(val)
            data_dict["wind"] = val_list

            for key in data_dict:
                try:
                    data_dict[key] = np.array(data_dict[key], dtype = float)
                except:
                    data_dict[key] = np.array(data_dict[key])
        df = (
            pd.DataFrame(data_dict)
            .drop(columns = ["Time",
                             "Weather",
                             "Sky Condition",
                             "Max Temp 6hr",
                             "Min Temp 6hr",
                             "Wind Chill",
                             "Heat Index",
                             "Other Pressure",
                             "Precip 3hr",
                             "Precip 6hr",
                             "Rel Humidity"])
        )
        ## replace missing values in precip_1hr with 0
        df["precip_1hr"] = df["precip_1hr"].replace("",0).astype(float)
        date_orderings = np.unique(data_dict["date"], return_index = True)
        idxs = np.argsort(np.argsort(date_orderings[1]))

        ## for all columns in df, check if there is a missing value, if so, fill it with the previous v
        for col in df.columns:
            try:
                df[col] = df[col].replace("", np.nan).fillna(method = "ffill").fillna(0)
            except:
                print("We are goners")

        ## necessary just in case the change of date happens....date seems to recorded in weather.gov a
        date_dict = {float(date): int(idxs[i]) for i, date in enumerate(date_orderings[0])}
        return df, date_dict

    def getCurrentWeatherData(self, place: str):
        today = datetime.today()
        hourly_df, date_orderings = self.getCurrenHourlyWeatherData(place)
        daily_df = (
        hourly_df
        .groupby('date')
        .agg({'temp': ['min', 'max', 'mean'],
            'dew_point': ['min', 'max', 'mean'],
```

```
            'precip_1hr': 'sum',
            'sea_level_pressure' : 'mean',
            'visibility': 'max',
            'wind': "max"})
        )
        daily_df.columns = ['_'.join(col) for col in daily_df.columns]
        ## removing grouping of daily_df
        daily_df = (
            daily_df
            .reset_index()
            .rename(columns = {'precip_1hr_sum': 'precipitation',
                               'sea_level_pressure_mean': 'sea_level_pressure',
                               'visibility_max': 'visibility',
                               'wind_max': 'max_wind_speed'})
        )

        date_range = pd.date_range(periods = len(daily_df), end = today, freq = "D")[::-1]
        daily_df["date"] = daily_df["date"].apply(lambda x: date_range[date_orderings[x]].floor("1d"))
        daily_df["location"] = place
        daily_df = daily_df.reindex(columns = ['location',
                                               'date',
                                               'temp_max',
                                               'temp_min',
                                               'temp_mean',
                                               'precipitation',
                                               'dew_point_max',
                                               'dew_point_min',
                                               'dew_point_mean',
                                               'max_wind_speed',
                                               'visibility',
                                               'sea_level_pressure'])
        return daily_df


    def predict_location(self, date, location: str):
        try:
            current_data = self.getCurrentWeatherData(location)
        except:
            print(f"Scraping Failed for {location}")
            current_data = None

        model = self.models[location]
        _ = model.predict(date, current_data)
        start_date = date - pd.Timedelta(days = 4)
        responses = (
            model.df[(model.df.date >= start_date) & (model.df.date <= date)]
            .copy()[["date", "temp_min", "temp_mean", "temp_max"]]
            .reset_index(drop = True)
            .sort_values(by = "date")
            .drop(columns = "date")
            .to_numpy()
        )
        return responses
```

```
    def predict_all(self):
        today = datetime.today().strftime("%Y-%m-%d")
        datetime_today = pd.to_datetime(today)
        datetime_end = datetime_today + timedelta(days = 5)

        locations_dict = {
            "Anchorage": "PANC",
            "Boise": "KBOI",
            "Chicago": "KORD",
            "Denver": "KCOS",
            "Detroit": "KDTW",
            "Honolulu": "PHNL",
            "Houston": "KIAH",
            "Miami": "KMIA",
            "Minneapolis": "KMSP",
            "Oklahoma City": "KOKC",
            "Nashville": "KBNA",
            "New York": "KJFK",
            "Phoenix": "KPHX",
            "Portland ME": "KPWM",
            "Portland OR": "KPDX",
            "Salt Lake City": "KSLC",
            "San Diego": "KSAN",
            "San Francisco": "KSFO",
            "Seattle": "KSEA",
            "Washington DC": "KDCA"
        }
        #order alphabetically
        locations = sorted(locations_dict.keys())
        #for location in locations:
        predictions = np.zeros((len(locations), 5, 3))
        for i, location in enumerate(locations):
            code = locations_dict[location]
            predictions[i,:,:] = self.predict_location(datetime_end, code)
        predictions = np.round(predictions, 1)
        print(f"{today}, {', '.join(str(ele) for ele in predictions.flatten())}")
        return predictions

## Example saving models for 1-7 lagged worth of lagged days
lag = 5
WFobj = WeatherForecast("data/", lag)
# save test
os.makedirs("models", exist_ok=True)
with open(f"models/WFobj_{lag}.pkl", "wb") as f:
    pickle.dump(WFobj, f)
```

7.3 Model Predict

```
# %%
import pandas as pd
import numpy as np
import os, pickle, janitor
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.gaussian_process import GaussianProcessRegressor as GPR
from sklearn.gaussian_process.kernels import RBF

import warnings

from warnings import simplefilter
simplefilter(action="ignore", category=pd.errors.PerformanceWarning)
simplefilter(action="ignore", category=FutureWarning)



## For the scraping
import requests
from bs4 import BeautifulSoup
from datetime import datetime, timedelta



# %%
def getCleanDataFrame(folder_path: str):
    files = os.listdir(folder_path)
    ## keep only the csv files
    files = [file for file in files if file.endswith(".csv")]
    df = pd.concat([pd.read_csv(folder_path + file)
                    .pipe(janitor.clean_names, strip_underscores=True)
                    .drop(columns=["unnamed_0"])
                    .assign(date=lambda x: pd.to_datetime(x.date),
                            temp_max=lambda x:pd.to_numeric(x.temp_max),
                            temp_min=lambda x:pd.to_numeric(x.temp_min),
                            temp_mean=lambda x:pd.to_numeric(x.temp_mean),
                            precipitation=lambda x:pd.to_numeric(x.precipitation),
                            dew_point_max=lambda x:pd.to_numeric(x.dew_point_max),
                            dew_point_min=lambda x:pd.to_numeric(x.dew_point_min),
                            dew_point_mean=lambda x:pd.to_numeric(x.dew_point_mean),
                            max_wind_speed=lambda x:pd.to_numeric(x.max_wind_speed),
                            visibility=lambda x:pd.to_numeric(x.visibility),
                            sea_level_pressure=lambda x:pd.to_numeric(x.sea_level_pressure)
                    )
                     for file in files])
    return df

# %%
## Created Lagged Data
def allLaggedDataAvail(df: pd.DataFrame, lag: int, to_lag_columns: list):
    df = df.sort_values(by=["date"])
    df['day_diff'] = df['date'].diff().dt.days
    df['has_all_lagged'] = (
        df['day_diff']
        .rolling(window=lag)
        .apply(lambda x: np.all(x == 1), raw=True)
    )
```

```
    for col in to_lag_columns:
        for i_lag in range(1, lag + 1):
            df[f'{col}_lag_{i_lag}'] = df[f'{col}'].shift(i_lag)
    df = df[df['has_all_lagged'] == 1].copy()
    df.dropna(subset=[f'{col}_lag_{lag}' for lag in range(1, lag + 1)], inplace=True)

    # Drop temporary columns
    df = df.drop(columns=['day_diff', 'has_all_lagged'])
    #df = df.drop(columns=to_lag_columns[3:])
    return df.sort_values(by = "date").reset_index(drop=True)

## Create a Model for each Location
class LocationModel():
    def __init__(self, location: str, df: pd.DataFrame, lag: int):
        self.to_lag_columns = ['temp_max',
                               'temp_min',
                               'temp_mean',
                               'precipitation',
                               'dew_point_max',
                               'dew_point_min',
                               'dew_point_mean',
                               'max_wind_speed',
                               'visibility',
                               'sea_level_pressure']
        self.lag = lag

        self.location = location
        self.df = df.copy()
        self.lagged_df = allLaggedDataAvail(df, lag, self.to_lag_columns)

    def fit(self):
        X_ = (
            self.lagged_df
            .drop(columns=self.to_lag_columns)
            .drop(columns = ["date", "location"])
            .to_numpy()
        )
        y_ = self.lagged_df[self.to_lag_columns].to_numpy()

        self.scaler = StandardScaler()
        X_ = self.scaler.fit_transform(X_)

        ## Define the Kernel
        gp = GPR(normalize_y=True, n_restarts_optimizer=10)

        ## Define the Parameters
        param_grid = {"alpha": np.logspace(1e-12,10,100),
                      "kernel": [RBF(l, length_scale_bounds="fixed") for l in np.logspace(-5,5,100)]}
        ## Define the Grid Search
        tscv = TimeSeriesSplit(n_splits=5)
        self.model = GridSearchCV(gp, param_grid=param_grid, cv=tscv, n_jobs=-1,
                                  scoring = "neg_mean_squared_error").fit(X_, y_)
        return self
```

```
def predict(self, date, scraped_data = None):
    ## If we are including the recent scraped data, append it into our data frame
    ## for better predictions
    if scraped_data is not None:
        bool_vec = np.zeros(len(scraped_data), dtype = bool)
        for i in range(len(scraped_data)):
            scraped_date = scraped_data.iloc[i]["date"]
            if scraped_date not in self.df["date"].values:
                bool_vec[i] = True
        self.df = pd.concat([self.df, scraped_data[bool_vec]])
        self.df = self.df.sort_values(by = "date")

    ## check if date is string
    if isinstance(date, str):
        date = pd.to_datetime(date)
    ## check if date is datetime
    if not isinstance(date, pd.Timestamp):
        raise ValueError("Date must be a string or a pd.Timestamp object")

    ## Clearly, if date is already present in dataframe, we can just return the stored values
    if date in self.df["date"].values:
        return self.df[self.df.date == date][self.to_lag_columns].to_numpy()
    else:
        ## check if last lagged days are in the dataframe
        prior_date_range = pd.date_range(end = date - pd.Timedelta(days=1), periods = self.lag + 1,
        for day in prior_date_range:

            if day not in self.df["date"].values:
                _ = self.predict(date=day)


        ## Now, we can predict the values for the date after previous lagged day values are present
        new_X_ = (
            self.df[(self.df.date >= prior_date_range[0]) & (self.df.date <= prior_date_range[-1])]
            .copy()
        )

        lagged_new_X = (
            allLaggedDataAvail(df = new_X_, lag = self.lag, to_lag_columns = self.to_lag_columns)
            .drop(columns = self.to_lag_columns)
            .drop(columns = ["date", "location"])
        )

        try:
            predicted_y = self.model.predict(self.scaler.transform(lagged_new_X.to_numpy()))[-1]
        except:
            ## fill missing values in lagged_new_X with 0
            lagged_new_X = lagged_new_X.fillna(0)
            predicted_y = self.model.predict(self.scaler.transform(lagged_new_X.to_numpy()))[-1]

        ## clip precipitation to be min 0
        if predicted_y[3] < 0:
```

```
                predicted_y[3] = 0

            ## clip visibility to be max 10
            if predicted_y[8] > 10:
                predicted_y[8] = 10

            ##
            predicted_obs = pd.DataFrame(predicted_y.reshape(1,10), columns = self.to_lag_columns)
            predicted_obs["date"] = date
            predicted_obs["location"] = self.location
            self.df = pd.concat([self.df, predicted_obs])
            self.df = self.df.sort_values(by = "date").reset_index(drop=True)
            return predicted_y

    def __str__(self):
        return f"{self.location} Model"


class WeatherForecast():
    def __init__(self, folder_path: str, lag: int):
        self.df = getCleanDataFrame(folder_path)
        self.locations = self.df.location.unique()
        self.models = {location: LocationModel(location, self.df[self.df.location == location], lag).fi

    def getCurrenHourlyWeatherData(self, place: str):
        col_names = ['date', 'Time', 'wind', 'visibility', 'Weather', 'Sky Condition', 'temp', 'dew_poi
                     'Min Temp 6hr', 'Rel Humidity', 'Wind Chill', 'Heat Index', 'sea_level_pressure', '
                     'precip_1hr', 'Precip 3hr', 'Precip 6hr']
        url = f"https://forecast.weather.gov/data/obhistory/{place}.html"

        response = requests.get(url)
        soup = BeautifulSoup(response.text, 'html.parser')
        table = soup.find('table', class_ = 'obs-history')
        for weather_data in table.find_all('tbody'):
            rows = weather_data .find_all('tr')
            data_dict = {col: [] for col in col_names}

            for row in rows:
                for col in col_names:
                    data_dict[col].append(row.find_all('td')[col_names.index(col)].text)
            val_list = []
            for i in data_dict["wind"]:
                val = i.split("\n")[1].replace(" ", "").replace("\n", "")
                if val == "":
                    val = "0"
                val_list.append(val)
            data_dict["wind"] = val_list

            for key in data_dict:
                try:
                    data_dict[key] = np.array(data_dict[key], dtype = float)
                except:
                    data_dict[key] = np.array(data_dict[key])
```

```
        df = (
            pd.DataFrame(data_dict)
            .drop(columns = ["Time",
                             "Weather",
                             "Sky Condition",
                             "Max Temp 6hr",
                             "Min Temp 6hr",
                             "Wind Chill",
                             "Heat Index",
                             "Other Pressure",
                             "Precip 3hr",
                             "Precip 6hr",
                             "Rel Humidity"])
        )
        ## replace missing values in precip_1hr with 0
        df["precip_1hr"] = df["precip_1hr"].replace("",0).astype(float)
        date_orderings = np.unique(data_dict["date"], return_index = True)
        idxs = np.argsort(np.argsort(date_orderings[1]))

        ## for all columns in df, check if there is a missing value, if so, fill it with the previous v
        for col in df.columns:
            try:
                df[col] = df[col].replace("", np.nan).fillna(method = "ffill").fillna(0)
            except:
                print("We are goners")

        ## necessary just in case the change of date happens....date seems to recorded in weather.gov a
        date_dict = {float(date): int(idxs[i]) for i, date in enumerate(date_orderings[0])}
        return df, date_dict

    def getCurrentWeatherData(self, place: str):
        today = datetime.today()
        hourly_df, date_orderings = self.getCurrenHourlyWeatherData(place)
        daily_df = (
        hourly_df
        .groupby('date')
        .agg({'temp': ['min', 'max', 'mean'],
            'dew_point': ['min', 'max', 'mean'],
            'precip_1hr': 'sum',
            'sea_level_pressure' : 'mean',
            'visibility': 'max',
            'wind': "max"})
        )
        daily_df.columns = ['_'.join(col) for col in daily_df.columns]
        ## removing grouping of daily_df
        daily_df = (
            daily_df
            .reset_index()
            .rename(columns = {'precip_1hr_sum': 'precipitation',
                               'sea_level_pressure_mean': 'sea_level_pressure',
                               'visibility_max': 'visibility',
                               'wind_max': 'max_wind_speed'})
        )
```

```
        date_range = pd.date_range(periods = len(daily_df), end = today, freq = "D")[::-1]
        daily_df["date"] = daily_df["date"].apply(lambda x: date_range[date_orderings[x]].floor("1d"))
        daily_df["location"] = place
        daily_df = daily_df.reindex(columns = ['location',
                                               'date',
                                               'temp_max',
                                               'temp_min',
                                               'temp_mean',
                                               'precipitation',
                                               'dew_point_max',
                                               'dew_point_min',
                                               'dew_point_mean',
                                               'max_wind_speed',
                                               'visibility',
                                               'sea_level_pressure'])
        return daily_df


    def predict_location(self, date, location: str):
        try:
            current_data = self.getCurrentWeatherData(location)
        except:
            print(f"Scraping Failed for {location}")
            current_data = None

        model = self.models[location]
        _ = model.predict(date, current_data)
        start_date = date - pd.Timedelta(days = 4)
        responses = (
            model.df[(model.df.date >= start_date) & (model.df.date <= date)]
            .copy()[["date", "temp_min", "temp_mean", "temp_max"]]
            .reset_index(drop = True)
            .sort_values(by = "date")
            .drop(columns = "date")
            .to_numpy()
        )
        return responses

    def predict_all(self):
        today = datetime.today().strftime("%Y-%m-%d")
        datetime_today = pd.to_datetime(today)
        datetime_end = datetime_today + timedelta(days = 5)
        print(self.locations)
        locations_dict = {
                "Anchorage": "PANC",
                "Boise": "KBOI",
                "Chicago": "KORD",
                "Denver": "KCOS",
                "Detroit": "KDTW",
                "Honolulu": "PHNL",
                "Houston": "KIAH",
                "Miami": "KMIA",
                "Minneapolis": "KMSP",
```

```
                "Oklahoma City": "KOKC",
                "Nashville": "KBNA",
                "New York": "KJFK",
                "Phoenix": "KPHX",
                "Portland ME": "KPWM",
                "Portland OR": "KPDX",
                "Salt Lake City": "KSLC",
                "San Diego": "KSAN",
                "San Francisco": "KSFO",
                "Seattle": "KSEA",
                "Washington DC": "KDCA"
            }
            #order alphabetically
        locations = sorted(locations_dict.keys())
        print(len(locations))
        print(len(self.locations))
        #for location in locations:
        predictions = np.zeros((len(locations), 5, 3))
        for i, location in enumerate(locations):
            code = locations_dict[location]
            try:
                predictions[i,:,:] = self.predict_location(datetime_end, code)
            except:
                print(f"Failed to predict {location} with code {code}")
        predictions = np.round(predictions, 1)
        print(f"{today}, {', '.join(str(ele) for ele in predictions.flatten())}")
        return predictions

# %%
lag = 5
try:
    with open(f"models/WFobj_{lag}.pkl", "rb") as f:
        wf = pickle.load(f)
except:
    print("No model found")
predictions = wf.predict_all()
```