Department of Information Engineering – MSc AIDE

Data Mining and Machine Learning – University of Pisa

Weather Prediction

Aida Himmiche
Michael Asante

Supervised by:
Prof. Francesco Marcelloni

ABSTRACT

The objective of this project is to build an application which operates in the domain of weather forecasting, in order to predict the weather in a certain city upon user demand. The concept of the application is to calculate the sought results relying on a trained predictive model and 5-year long history records of temperature, humidity, pressure, wind direction and wind speed for 35 cities in 3 countries. The data was also collected on an hourly basis, which will allow the user to refine their search and input time parameters to get the temperature in degree celsius.

The primary aim of this project was to predict the temperature values for one of the cities present in the dataset at a certain date and time. The future work would be to also predict the other weather attributes (humidity, pressure, wind direction, wind speed, weather) for the chosen city at a certain date and time, for which a feasibility study and primary sketch were made.

This application was developed using Machine Learning(ML). An analysis of various models was conducted, then the best predictive model was trained and tested on the dataset for the best accuracy achievable. Moreover, for the sake of applying the material covered in the Data Mining and Machine Learning course and truly evaluating our model, an analysis using the Cross-Validation methodology was carried out.

Keywords: *Weather prediction, Machine Learning, Cross-validation, Predictive model.*

TABLE OF CONTENTS

KDD Process

The dataset

The dataset used consists of data from two sources, NASA and Renewables.ninja. Both were solicited to collect weather records on 36 cities (*Vancouver, Portland, San Francisco, Seattle, Los Angeles, San Diego, Las Vegas, Phoenix, Denver, San Antonio, Dallas, Houston, Kansas City, Minneapolis, Saint Louis, Chicago, Nashville, Indianapolis, Atlanta, Detroit, Jacksonville, Charlotte, Miami, Pittsburgh, Toronto, Philadelphia, New York, Montreal, Boston, Beersheba, Tel Aviv District, Eilat, Haifa, Nahariyya, Jerusalem*) from 3 countries (*USA, Canada, Israel*). Data was collected each day for every hour, in a time period of 5 years, thus from 2012 to 2017.

Notes on the dataset:

- The original formatting was not ideal: The initial data was split into 6 Comma-Separated-Values (CSV) files, organized by City name versus Datetime(the data collection), dedicated to each feature:
    - Temperature
    - Humidity
    - Pressure
    - Wind Direction
    - Wind Speed

- Clamped attributes: The "datetime" data was given as a string of "YYYY-MM-DD hh:mm", with the full date and time in the same field.

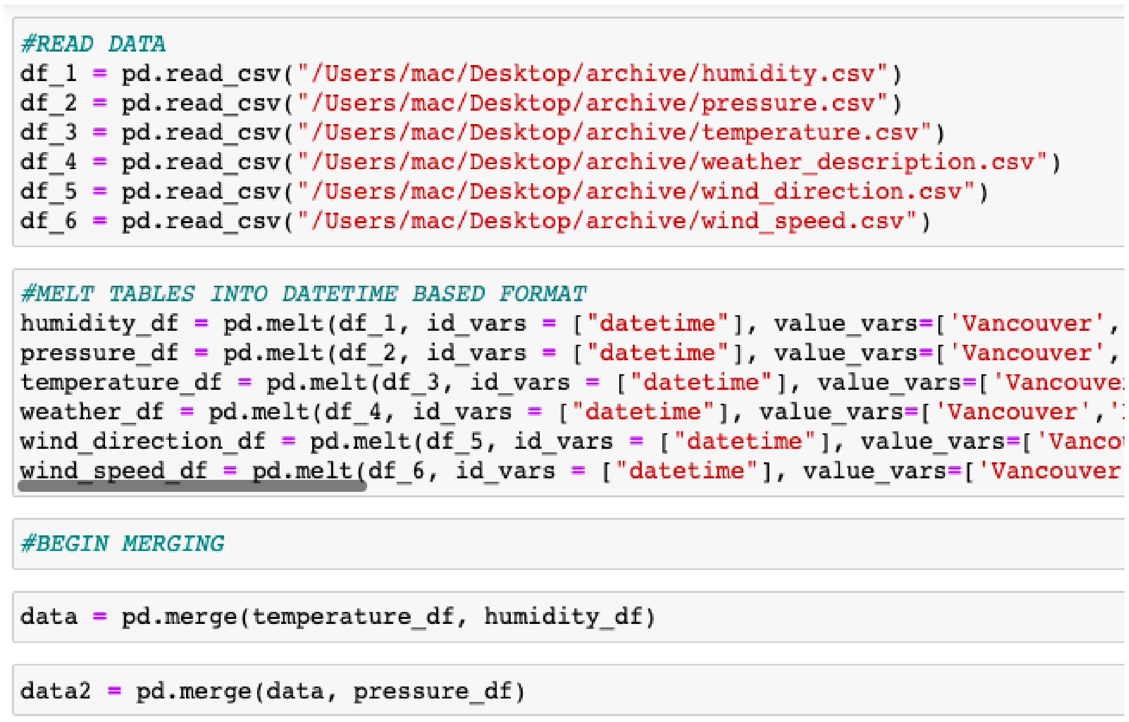
- For 9 out of 36 cities, the hourly data collection stopped one month earlier than the rest. In addition to some missing values through the dataset, 34 days worth of hourly records were missing from 28-10-2017 to 31-11- 2017 in the following cities:
    - Vancouver
    - San Francisco
    - Pittsburgh
    - Montreal
    - Tel Aviv
    - Eilat
    - Haifa
    - Nahariyya
    - Jerusalem

Preprocessing

To deal with some of the notes we took on the dataset, we carried out some preprocessing tasks. First, we needed to reformat our datasets by merging them into one complete file. We used Pandas for most of the data-related steps, relying on built-in methods and Data Frame object structures.

Data Integration

To format and merge the files we made use of the method *“pd.melt()”* to reorganize our individual tables according to the Datetime attribute, and introduced the cities as a column instead of an x-axis row. Once all the files had the same shape, we used the method *“pd.merge()”* to join each feature column to the rest and end up with one full table.



```
#READ DATA
df_1 = pd.read_csv("/Users/mac/Desktop/archive/humidity.csv")
df_2 = pd.read_csv("/Users/mac/Desktop/archive/pressure.csv")
df_3 = pd.read_csv("/Users/mac/Desktop/archive/temperature.csv")
df_4 = pd.read_csv("/Users/mac/Desktop/archive/weather_description.csv")
df_5 = pd.read_csv("/Users/mac/Desktop/archive/wind_direction.csv")
df_6 = pd.read_csv("/Users/mac/Desktop/archive/wind_speed.csv")
```

```
#MELT TABLES INTO DATETIME BASED FORMAT
humidity_df = pd.melt(df_1, id_vars = ["datetime"], value_vars=['Vancouver',
pressure_df = pd.melt(df_2, id_vars = ["datetime"], value_vars=['Vancouver',
temperature_df = pd.melt(df_3, id_vars = ["datetime"], value_vars=['Vancouve
weather_df = pd.melt(df_4, id_vars = ["datetime"], value_vars=['Vancouver','
wind_direction_df = pd.melt(df_5, id_vars = ["datetime"], value_vars=['Vanco
wind_speed_df = pd.melt(df_6, id_vars = ["datetime"], value_vars=['Vancouver
```

```
#BEGIN MERGING
```

```
data = pd.merge(temperature_df, humidity_df)
```

```
data2 = pd.merge(data, pressure_df)
```

The integrated transposed raw dataset had 1629108 rows and 8 attribute fields. The unique columns which ran through the datasets were the City and Datetime columns which became the merging points. A cross-section of the dataset highlights the following fields and their data types.

Column Heading	**Type**	**Description**
datetime	object	A description for date and time for which the recordings were collected.
Country	object	Although the column is made up of Cities, it was represented as a Country column highlighting the cities for collected data.
temperature	float	The recordings of temperature measured in Kelvin(K). Min recording is 242.336666667 and Max being 321.22
wind_direction	object	The state for wind recorded at the time and day for a city
wind_speed	float	The speed recorded at that time for a city and a time.
pressure	float	The pressure reading corresponds to a city and a time.
humidity	float	Humidity recordings which ranges from 100(max) to 5(min)

Data Cleaning

Regarding the missing values, we dealt with the issue in two ways:

- **Dropping the missing data**: since 34 days of hourly data wasn't collected for 9 cities, we decided to drop the time period from 28-10-2017 to 31-11-2017 for all cities, even those that did hold data. We used the *"DataFrame.drop()"* method by providing an array of indices that matched the time frame.
- **Replacing the missing values**: For the missing data throughout the dataset, we opted to replace all the None values with the modes of each column. The function to fill in the Null values was the Linear forward cross interpolation which considered values to the 3 length close to the datapoint to fill the value.

```
weather_df_noNov.interpolate(method ='linear', limit_direction ='forward', inplace=True)
```

```
for i in weather_df_noNov.columns:
    weather_df_noNov[i].fillna(weather_df_noNov[i].mode()[0], inplace=True)
```

Data Transformation

Feature Encoding

Since the temperature prediction will be a Predictive Machine Learning model, the transformed dataset represents the information of a weather prediction reading, picking some attributes to train the machine learning model. For example, the attribute *City* is remapped as a categorical attribute.

Considering the transformation to be applied, the "*One Hot Encoding"* technique was used in order to transform all the nominal attributes into numerical form. This method re-oriented the dataset by bringing in other fields for which the City value for that particular row was 1 and all other City columns turned into 0. Using a different encoder, eg. Integer encoding will make the model assume a natural ordering between categories and it may result in a poor performance or giving unexpected results.
After the One Hot Encoding, the shape of the data was (1287238 rows , 40 columns)
Our dataset contained a Nominal Categorical attributes:

- **City:** this categorical attribute contains 35 distinct values, and to encode it we chose to use the One-Hot Encoding scheme which relies on binary representation and creates a new column for every categorical value, then assigns a "1" to rows that contained that value, or a "0" to the ones that didn't.

Feature Reduction

In order to make computational work faster and easier, an analysis for Correlation together with the Spearman Correlation function was checked which highlighted some features which will not affect computation and improve model efficiency. These included:

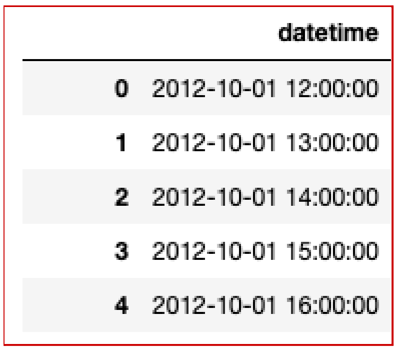
- **Minute:** The attribute being at a constant 0 makes its variance null at all times, and therefore gives an undefined correlation calculation. It is then useless in the prediction scheme as it never changes for any record, therefore, it was dropped.

Feature Extraction/ Construction

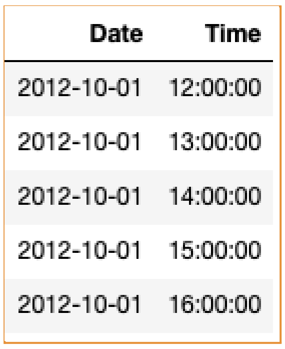
To deal with the DateTime attribute which contained the entire date and the time of the data collection, we used the *"str.split()"* method and provided the delimiters to take into account, as well as the new columns in which to store the separate data.

```
weather_df_split_time = weather_df_split_datetime
weather_df_split_time [['Hour','Minute','Second']] = weather_df_split_datetime['Time'].str.split(':',expand=True)
```

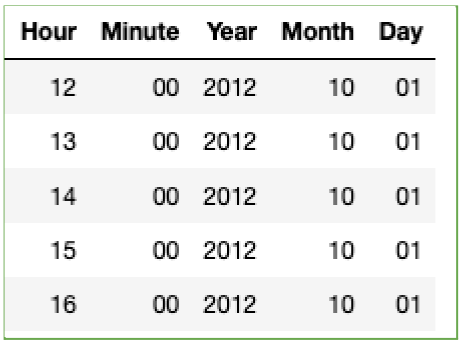
```
weather_df_split_date = weather_df_split_time
weather_df_split_date [['Year','Month','Day']] = weather_df_split_time['Date'].str.split('-',expand=True)
```



	datetime
0	2012-10-01 12:00:00
1	2012-10-01 13:00:00
2	2012-10-01 14:00:00
3	2012-10-01 15:00:00
4	2012-10-01 16:00:00

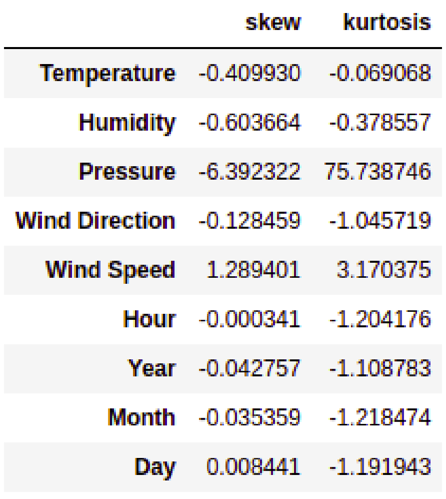


Date	Time
2012-10-01	12:00:00
2012-10-01	13:00:00
2012-10-01	14:00:00
2012-10-01	15:00:00
2012-10-01	16:00:00



Hour	Minute	Year	Month	Day
12	00	2012	10	01
13	00	2012	10	01
14	00	2012	10	01
15	00	2012	10	01
16	00	2012	10	01

Afterwards, to further identify the important attributes that will improve the Model performance and avoid misleading results, we performed some feature extraction techniques based on the attribute fields for the dataset.



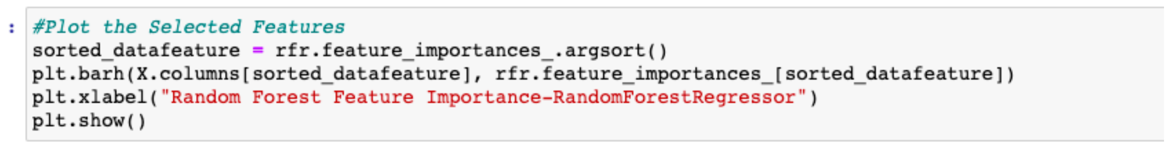
	skew	kurtosis
Temperature	-0.409930	-0.069068
Humidity	-0.603664	-0.378557
Pressure	-6.392322	75.738746
Wind Direction	-0.128459	-1.045719
Wind Speed	1.289401	3.170375
Hour	-0.000341	-1.204176
Year	-0.042757	-1.108783
Month	-0.035359	-1.218474
Day	0.008441	-1.191943

The figurative representation of Skewness and Kurtosis was studied to understand the symmetry of the Dataset as well as how heavy/light tailed the data is. Thus, showing some features that we can apply some transformation techniques to. As a general rule of thumb, skewness can be interpreted in this way:

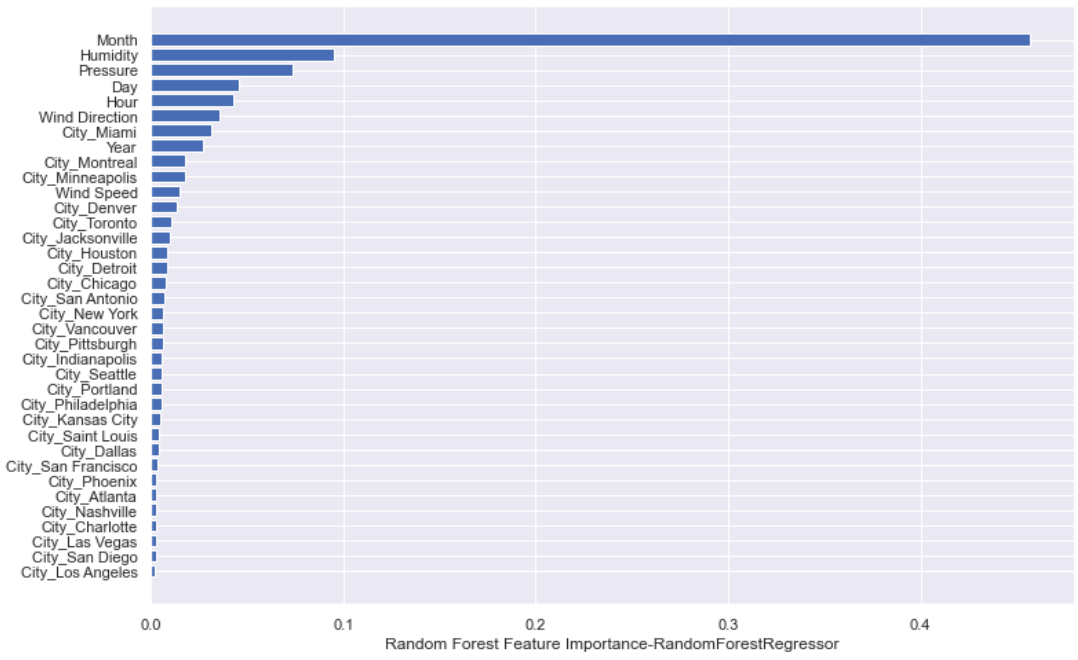
Fairly Symmetrical	-0.5 to 0.5
Moderate Skewed	-0.5 to -1.0 and 0.5 to 1.0
Highly Skewed	< -1.0 and > 1.0

1. Extracting Day, Month, Year and Hour

   The weather data time-stamp in its raw format does not convey much information. Since the data was collected on a cross-section of hours, the minute of the time-stamp will not correlate to the other data attributes. Also considering the granularity of the dataset, RandomForestRegressor was used to understand the important features within the data and sorted based on the level of relevance in our model performance and prediction.

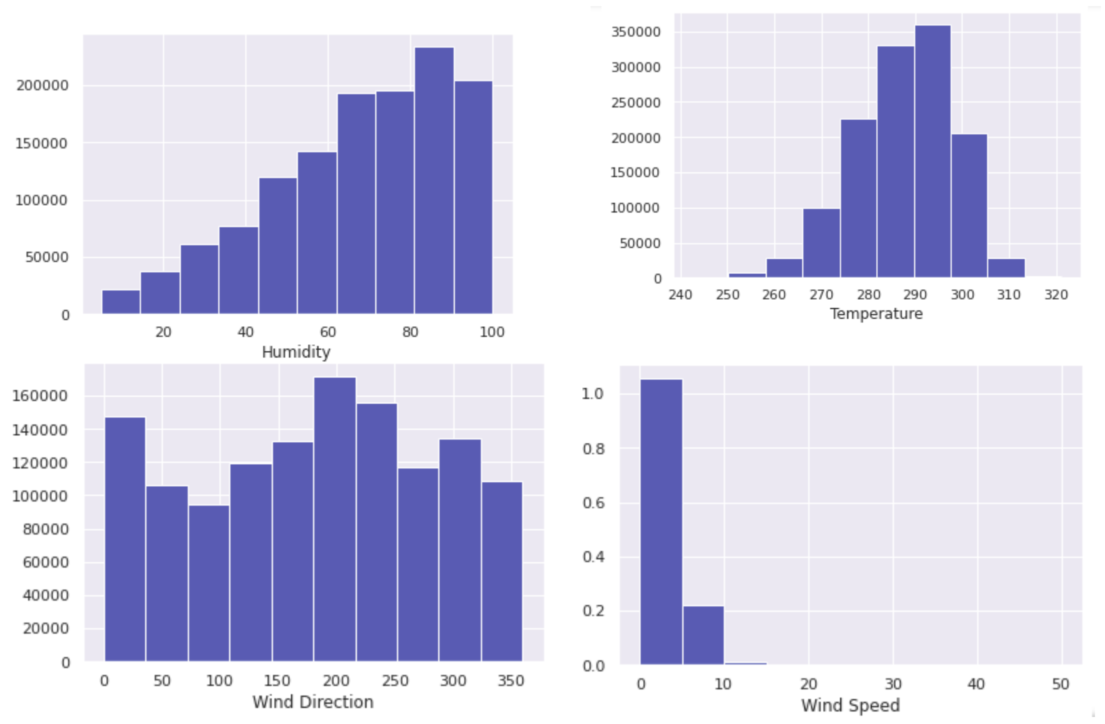


```
#Plot the Selected Features
sorted_datafeature = rfr.feature_importances_.argsort()
plt.barh(X.columns[sorted_datafeature], rfr.feature_importances_[sorted_datafeature])
plt.xlabel("Random Forest Feature Importance-RandomForestRegressor")
plt.show()
```



2. Statistical Transformation

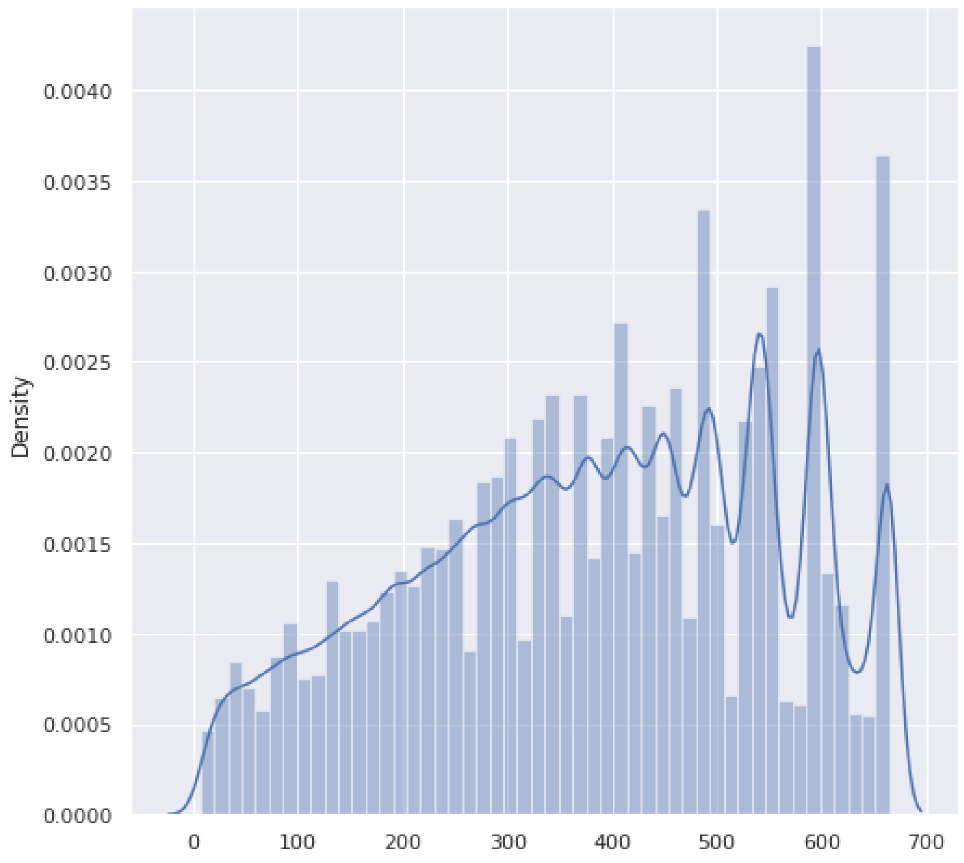
   Observing certain columns and their skewness within the Dataset, of which some are numerical values, certain observations and conclusions were drawn on them to decide which statistical feature was computed on them. A cross-section of skewness which shows the spread of the data features are represented below.



An overview of the above dataset showed the distribution of some data points within the dataset and how skewed some of the features were. To perform normalization, the Box Cox transformation technique was used.

Box Cox transformation

Looking at the features being positive, and not normally distributed with features not having a constant variance, the Box Cox transformation was implemented. Using the scipy stats function, box cox transformation was applied on the Humidity and Pressure column (one-dimensional) which highlighted an improved distribution close to a normal distribution curve



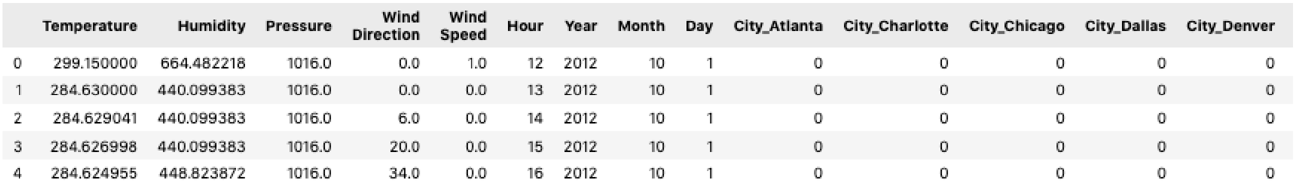
After data transformation, there was an improved distribution of the data analyzing the skewness and kurtosis of the dataset.

	skew	kurtosis
Temperature	-0.409930	-0.069068
Humidity	-0.253706	-0.878564
Pressure	-6.392322	75.738746
Wind Direction	-0.128459	-1.045719
Wind Speed	1.289401	3.170375
Hour	-0.000341	-1.204176
Year	-0.042757	-1.108783
Month	-0.035359	-1.218474
Day	0.008441	-1.191943

Final Dataset

The final dataset after the preprocessing steps were carried out:

Column Heading	**Type**	**Description**
Year	Integer	The year of the record.
Month	Integer	The month of the record.
Day	Integer	The day of the record.
Hour	Integer	The hour of the record.
Temperature	float	The recordings of temperature measured in Kelvin(K). Min recording is 242.336666667 and Max being 321.22
Humidity	float	Humidity recordings which ranges from 100(max) to 5(min)
Pressure	float	The pressure reading corresponds to a city and a time.
Wind Speed	float	The speed recorded at that time for a city and a time.
Wind Direction	float	The state for wind recorded at the time and day for a city
City_Atlanta	Boolean	Dummy column: 1 if it is the city in the record, 0 if not.
...	...	...
City_Vancouver	Boolean	Dummy column: 1 if it is the city in the record, 0 if not.



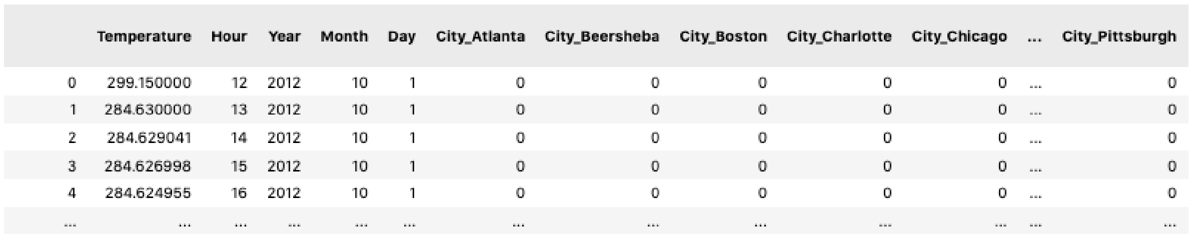
	Temperature	Humidity	Pressure	Wind Direction	Wind Speed	Hour	Year	Month	Day	City_Atlanta	City_Charlotte	City_Chicago	City_Dallas	City_Denver
0	299.150000	664.482218	1016.0	0.0	1.0	12	2012	10	1	0	0	0	0	0
1	284.630000	440.099383	1016.0	0.0	0.0	13	2012	10	1	0	0	0	0	0
2	284.629041	440.099383	1016.0	6.0	0.0	14	2012	10	1	0	0	0	0	0
3	284.626998	440.099383	1016.0	20.0	0.0	15	2012	10	1	0	0	0	0	0
4	284.624955	448.823872	1016.0	34.0	0.0	16	2012	10	1	0	0	0	0	0

Data Mining

This section is dedicated to the work done on predicting the **<u>Temperature</u>** values for the cities in the dataset, upon a user input of date and time. For this step of the KDD process we mainly used Python's Scikit-learn library, which is a simple and effective tool for predictive data analysis.

Since the user wouldn't be inputting any information on the Humidity, Pressure, Wind direction or speed, we needed to predict the temperature solely based on the date, time, and location.

Therefore, the dataset used for this numerical value prediction was the following:



	Temperature	Hour	Year	Month	Day	City_Atlanta	City_Beersheba	City_Boston	City_Charlotte	City_Chicago	...	City_Pittsburgh
0	299.150000	12	2012	10	1	0	0	0	0	0	...	0
1	284.630000	13	2012	10	1	0	0	0	0	0	...	0
2	284.629041	14	2012	10	1	0	0	0	0	0	...	0
3	284.626998	15	2012	10	1	0	0	0	0	0	...	0
4	284.624955	16	2012	10	1	0	0	0	0	0	...	0
...	...	...	...	...	...	...	...	...	...	...	...	...

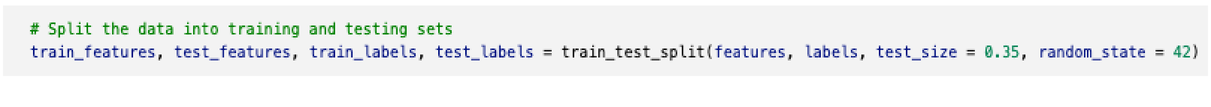
Train-Test Split

Firstly, we separated our labels (data to be predicted: Temperature) and the features (the rest of the attributes) in different variables.

The "sklearn" methods we used to build our models take data in arrays, so we transformed our temperature dataset into such.

Then, we needed to split the dataset into training data; the labeled data that the model will use during the learning process in order to find the necessary patterns, and testing data; the labeled data that it will test its own performance against.

We used Python's "train_test_split()" from the library sklearn with a train-test ratio of 65%-35% respectively.



```
# Split the data into training and testing sets
train_features, test_features, train_labels, test_labels = train_test_split(features, labels, test_size = 0.35, random_state = 42)
```

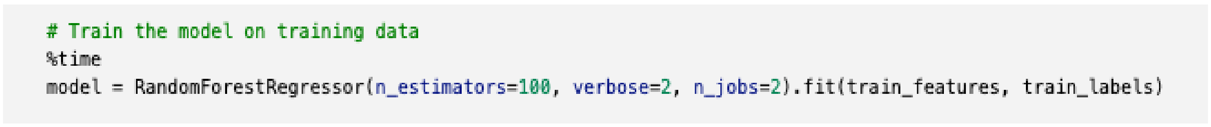
Predictive model algorithms:

To choose which Machine Learning model we would integrate in our application, we tried different classification and prediction algorithms to predict our temperature values. We started by studying their requirements and purposes, then ruled some of them out, and kept the others as options.

Random Forest Model

This first model is a supervised classification/regression model which builds decision trees on samples of provided data. Sklearn's RandomForestRegressor uses the Bagging method for this task, creating different training subsets and deciding on the final output based on average votes (Majority for classification).

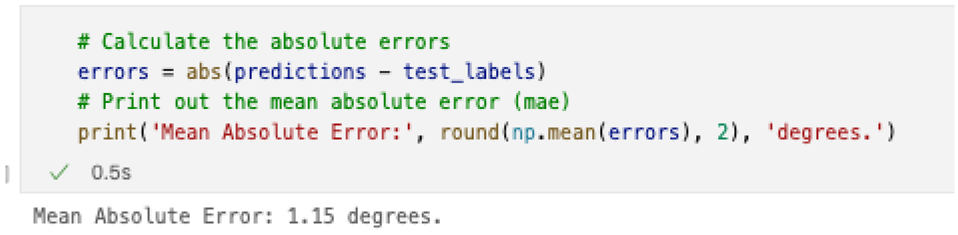
We imported the Random Forest model from skLearn and instantiated it to generate 100 trees, then fit it to our training features and labels. The training took 23.5 minutes, with parallelization of two concurrent workers in ThreadingBackend.



```
# Train the model on training data
%time
model = RandomForestRegressor(n_estimators=100, verbose=2, n_jobs=2).fit(train_features, train_labels)
```

Evaluation

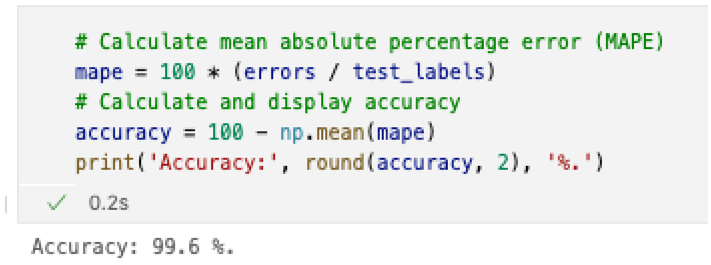
We used a few evaluation methods for our Random Forest model. Starting with a simple accuracy score, we calculated the Mean Absolute Error of the predictions, which was a low 1.15 degrees, then used the number of errors to calculate a Mean Absolute Percentage Error (MAPE) that allows us to find the accuracy out of 100.



```
# Calculate the absolute errors
errors = abs(predictions - test_labels)
# Print out the mean absolute error (mae)
print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')
```

✓ 0.5s

```
Mean Absolute Error: 1.15 degrees.
```



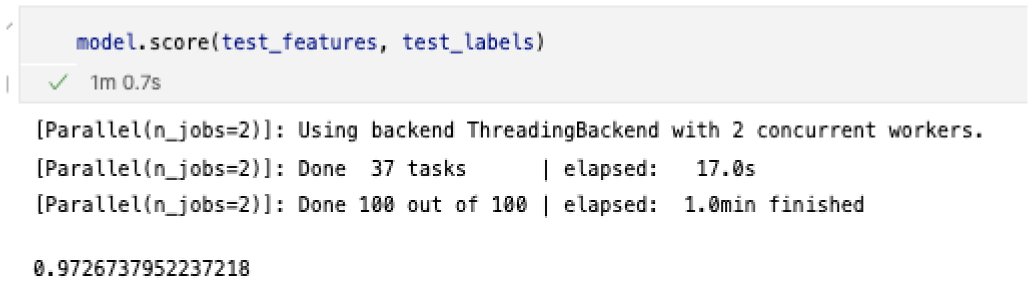
```
# Calculate mean absolute percentage error (MAPE)
mape = 100 * (errors / test_labels)
# Calculate and display accuracy
accuracy = 100 - np.mean(mape)
print('Accuracy:', round(accuracy, 2), '%.')
```

✓ 0.2s

```
Accuracy: 99.6 %.
```

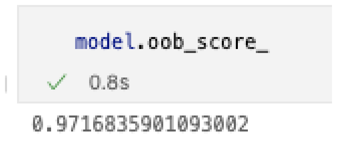
From this previous method we can see that the accuracy of the model is said to be a high 99.6%, but this metric is (more often than not) unreliable, therefore we also tried another evaluation method.

According to the score() the model has a 97% accuracy against unseen data:



```
model.score(test_features, test_labels)
✓ 1m 0.7s
[Parallel(n_jobs=2)]: Using backend ThreadingBackend with 2 concurrent workers.
[Parallel(n_jobs=2)]: Done  37 tasks      | elapsed:   17.0s
[Parallel(n_jobs=2)]: Done 100 out of 100 | elapsed:  1.0min finished

0.9726737952237218
```

Another evaluation method is Out-Of-Bag score, a metric used for Random Forest models and other which use “boostrap aggregating/bagging”, the results converge with 97%:



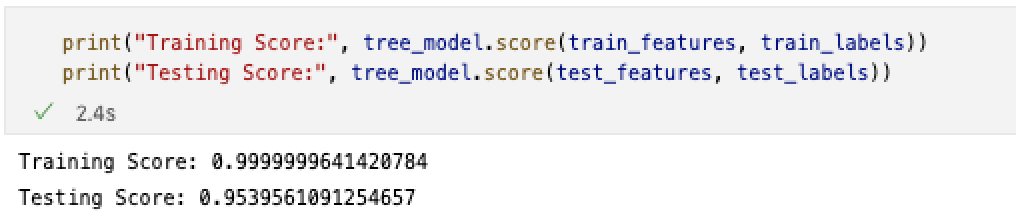
```
model.oob_score_
✓ 0.8s
0.9716835901093002
```

Decision Tree

Our next option was another supervised learning method: the Decision Tree model, built with the DecisionTreeRegressor() from sklearn. This algorithm creates mapped trees which illustrate all possible outcomes of a regression/classification task using a branching system to find the final output.

We fit this model to our training data and predicted the values of the testing set. The training took 23s for this model.

Evaluation

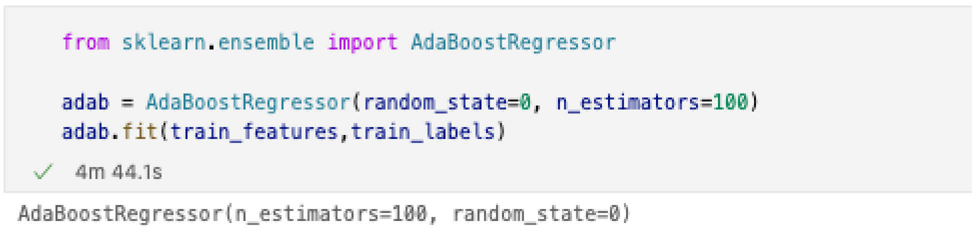


```
print("Training Score:", tree_model.score(train_features, train_labels))
print("Testing Score:", tree_model.score(test_features, test_labels))
✓ 2.4s
```

```
Training Score: 0.9999999641420784
Testing Score: 0.9539561091254657
```

AdaBoost

For AdaBoost, the training took 4minutes in total.

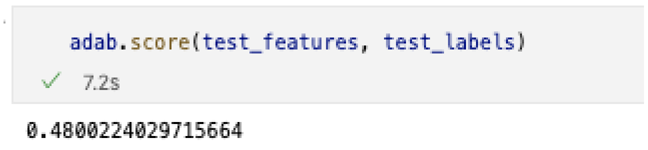


```
from sklearn.ensemble import AdaBoostRegressor

adab = AdaBoostRegressor(random_state=0, n_estimators=100)
adab.fit(train_features,train_labels)
✓ 4m 44.1s
```

```
AdaBoostRegressor(n_estimators=100, random_state=0)
```

Evaluation

The score for this model was very low, down to 48% only. What we can conclude from this attempt is that the boosting method may not be appropriate for this task.

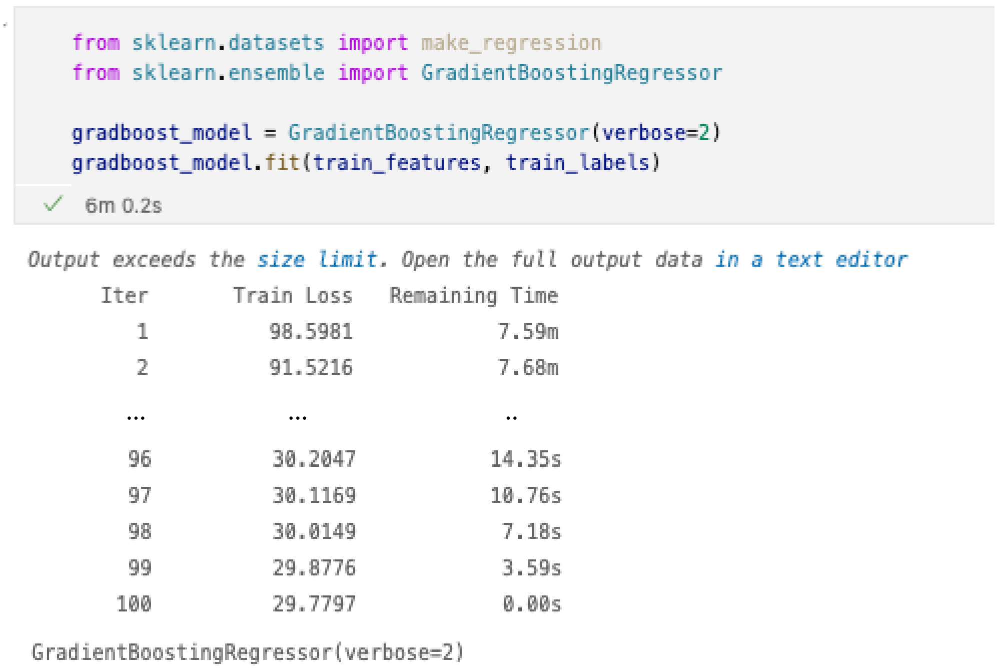


```
adab.score(test_features, test_labels)
✓ 7.2s
```

```
0.4800224029715664
```

Gradient Boosting

The generalization of AdaBoosting is Gradient Boosting. It is meant to minimize the loss of Adaboost models by adding weak learners progressively in a gradient-like way. The training took 6 minutes to complete.



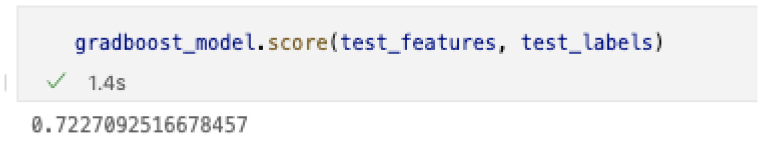
```
from sklearn.datasets import make_regression
from sklearn.ensemble import GradientBoostingRegressor

gradboost_model = GradientBoostingRegressor(verbose=2)
gradboost_model.fit(train_features, train_labels)
```

✓ 6m 0.2s

```
Output exceeds the size limit. Open the full output data in a text editor
      Iter       Train Loss   Remaining Time
         1          98.5981            7.59m
         2          91.5216            7.68m

        ...          ...               ..

        96          30.2047           14.35s
        97          30.1169           10.76s
        98          30.0149            7.18s
        99          29.8776            3.59s
       100          29.7797            0.00s

GradientBoostingRegressor(verbose=2)
```

Evaluation

This model's accuracy was higher than AdaBoost with 72%, which means it succeeded at minimizing the loss, but it remains worse in terms of performance than the first two models.



```
gradboost_model.score(test_features, test_labels)
```

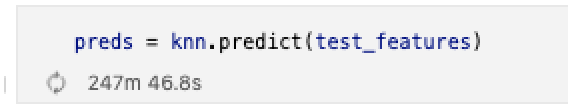
✓ 1.4s

```
0.7227092516678457
```

K Nearest Neighbors (KNN)

This supervised machine learning algorithm is simple and can be used to solve regression problems such as in our case. It works by finding the distance between the inputted sample values and examples throughout the dataset, it then collects the closest examples to the query and assumes an average vote to assign the output.

There is practically no training involved in building this model, however, making predictions is when the model slows down considerably. The larger the data to be predicted, the slower the model works, and in our case since we had over 50000 samples to predict during evaluation, the execution time was endless.



```
preds = knn.predict(test_features)
247m 46.8s
```

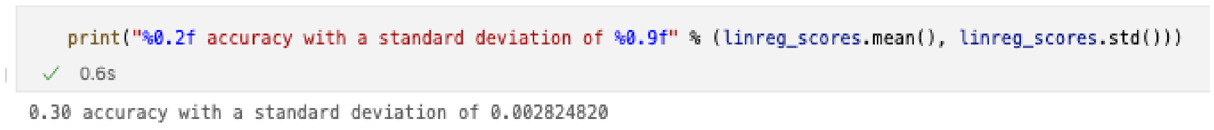
It was taking up to 247minutes on 35% of our dataset (testing set), an equivalent of 4 hours. We unfortunately could not continue running and therefore could not evaluate it against the other models we built.

Learning methods that couldn't be used

Linear Regression

Linear regression is a simple model that predicts the relationship between two or more variable values. This model predicts one value of one variable based on the value of another (depending/dependent values scheme).

This model would be useful if only our data was linear as the name suggests. We tried it anyway just to prove a point, and got an accuracy of 30%, which is the lowest score so far.



```
print("%0.2f accuracy with a standard deviation of %0.9f" % (linreg_scores.mean(), linreg_scores.std()))
✓ 0.6s
0.30 accuracy with a standard deviation of 0.002824820
```

AutoRegressive (AR)

This is a time-series based learning method which assumes that the data follows a certain time sequence, and uses data points at previous time steps to predict those as the next time steps with a regression equation. This model cannot be used in our case because although our data follows a time based progression, it is collected in different cities, with the time sequence repeating throughout the dataset.

All time-series algorithms (Autoregressive Integrated Moving Average (ARIMA), Seasonal Autoregressive Integrated Moving Average (SARIMA), Exponential Smoothing (ES), XGBoost, Prophet, LSTM (Deep Learning), DeepAR, N-BEATS, Temporal Fusion Transformer (Google)) were then ruled out.

Models benchmark

Model	Score	Total Time
Random Forest Regressor	97%	25 minutes
Decision Tree Regressor	95%	23 seconds
AdaBoost	48%	4 minutes
Gradient Boosting Regressor	72%	6 minutes
K Nearest Neighbors (KNN)	-	4 hours +

Looking at the accuracy score versus the time it took to train the models, the answer is torn between Random Forest and Decision Tree. If time was no problem, Random forest is a better pick, otherwise the Decision Tree model has an acceptable/high accuracy as well.

However, for us the issue of accuracy is very important because predicting temperature is only one task amongst those we would wish to perform. The next section will explain this concern, and why we chose **Random Forest** as the final model to integrate in our application.

Future Work

Predicting temperature is the first task of weather prediction in this project. Granted it is the only task we implemented in our application, but the bigger idea is to predict all 5 values and show them to the user when they are asking for a weather forecast at a certain city, during a date and time.

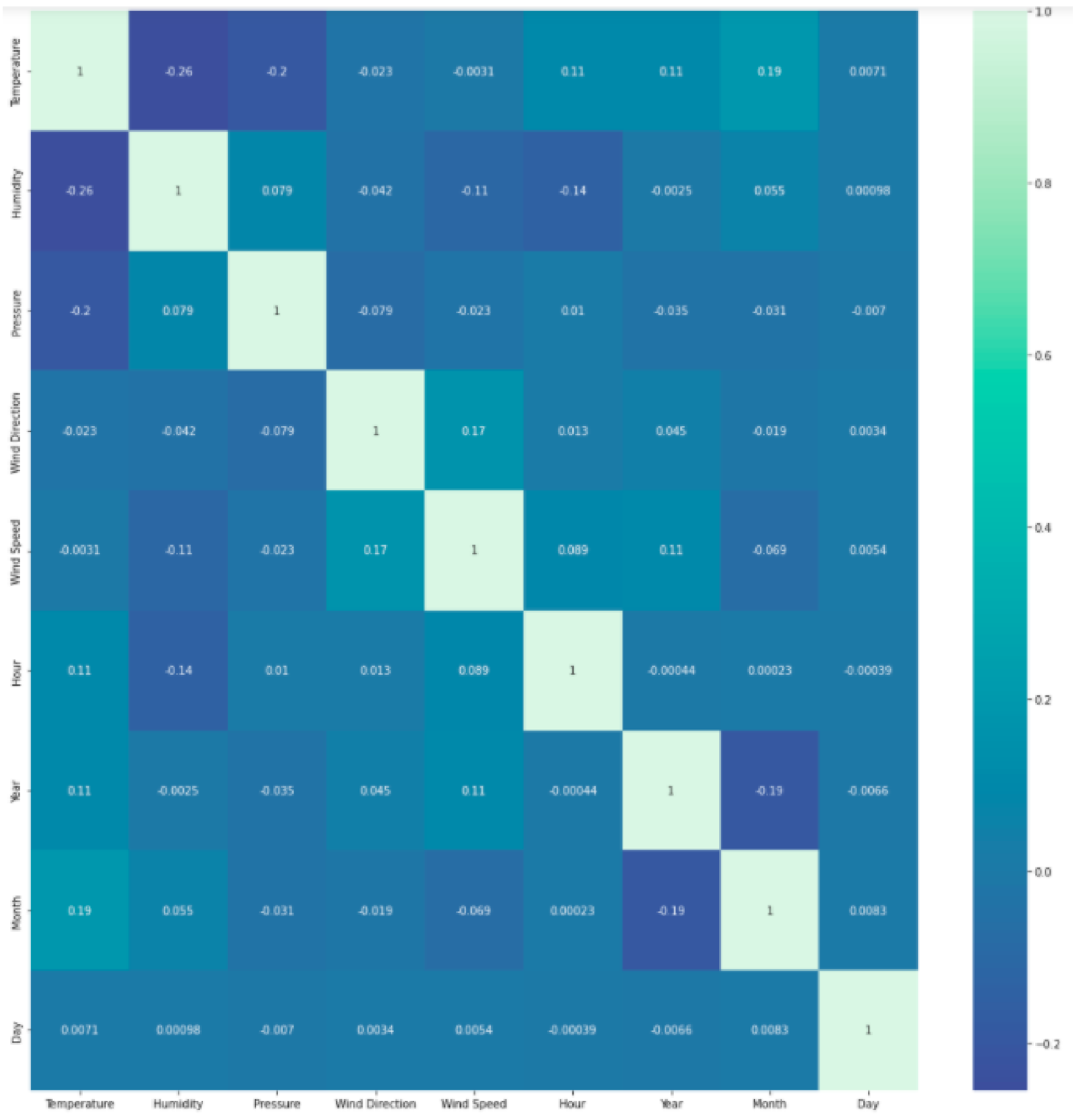
Since the user will only input date, time and location, it wouldn't be possible to predict the rest of the values using one model. The solution we thought of is using the temperature prediction as input and building somewhat of a waterfall architecture between models.

The way this will work is by making each prediction an input for another model, Starting with temperature as it is the most basic information to provide a user.

Feature Selection

Deciding on which feature to predict next is crucial for this type of task. With the temperature being predicted at an accuracy of 97%, we need to make sure the next attribute is predicted as closely as possible to the actual value, so that the accuracy doesn't drop lower and the error rate doesn't accumulate greatly after every step.

To deal with this problem, we are going to perform some feature selection to know which attribute matters the most when predicting another. An overview on this matter could be provided with a correlation matrix:



The correlation plot shows a negative correlation between Humidity and Temperature (-0.26), as well as Pressure and Temperature(-0.2). In this case we can choose to predict Humidity first since the patterns may be slightly more obvious, followed by the Pressure since it also has a positive correlation with Humidity (0.076). After that, we could predict Wind Speed which has a negative correlation with Humidity (-0.11), then the Wind Direction that is correlated positively with the speed and negatively with Pressure(-0.079)

We can also observe negative correlation between the Hour and the humidity, suggesting that the lower the “Hour” value (typically early in the morning) the Humidity level is the highest.

The use of Attribute Selection algorithms may give more precise results, but correlation gives a basic idea of how to proceed.

The models for each of these predictions will also be Random Forest models for the accuracy issue mentioned before, but the reason we didn’t implement them was because of computing capacity/storage issues.

Application

Analysis Phase

Description

The application is designed for meteorologists and people interested in predicting temperature values before stepping out. A user can navigate the application by entering values of Date, Time and City and click the Predict button for which the user gets the temperature value.

Main Actors

There is only one actor: *generic user*. He/she is the person who wants to predict the temperature value. There is no registration phase and no other information related to the generic user is needed.

Scenariosishere is only one scenario for the Weather Prediction application:

The Scenario for the Weather Prediction shows a user inserting values from a Date and Time Picker, selects a country through which he/she gets the City to choose. This information is the same which was analyzed during the training of the model.

Requirements

Functional Requirements

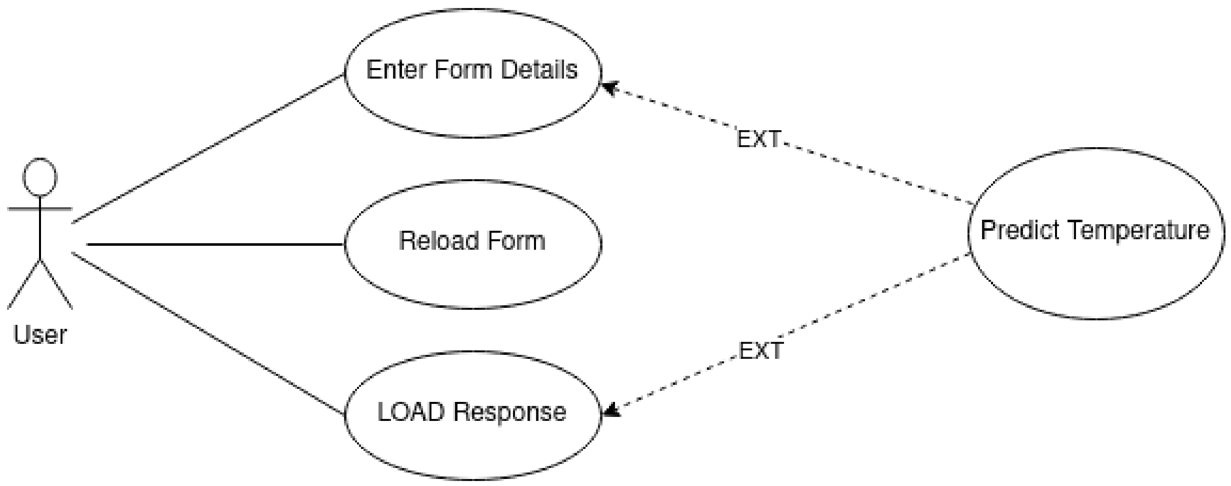
The application must relate to the information entered by a user to predict the temperature. The application consumes an endpoint API through the user's entered information, simplifying it to submit to the model. After the details are submitted and the temperature value predicted, the application can be refreshed to re-enter information again.

Non-Functional Requirements

The application has a simple GUI which enables the user to easily navigate, portable and intuitive. The application has an embedded ML classification prediction to make a prediction. The user should not wait too much time to get a prediction for the values entered. The model should predict results which are not misleading and also be able to adapt well to unseen data and also be able to learn from it. The code is easily composed and should be able to add future functionalities to it.

Use Case Diagram

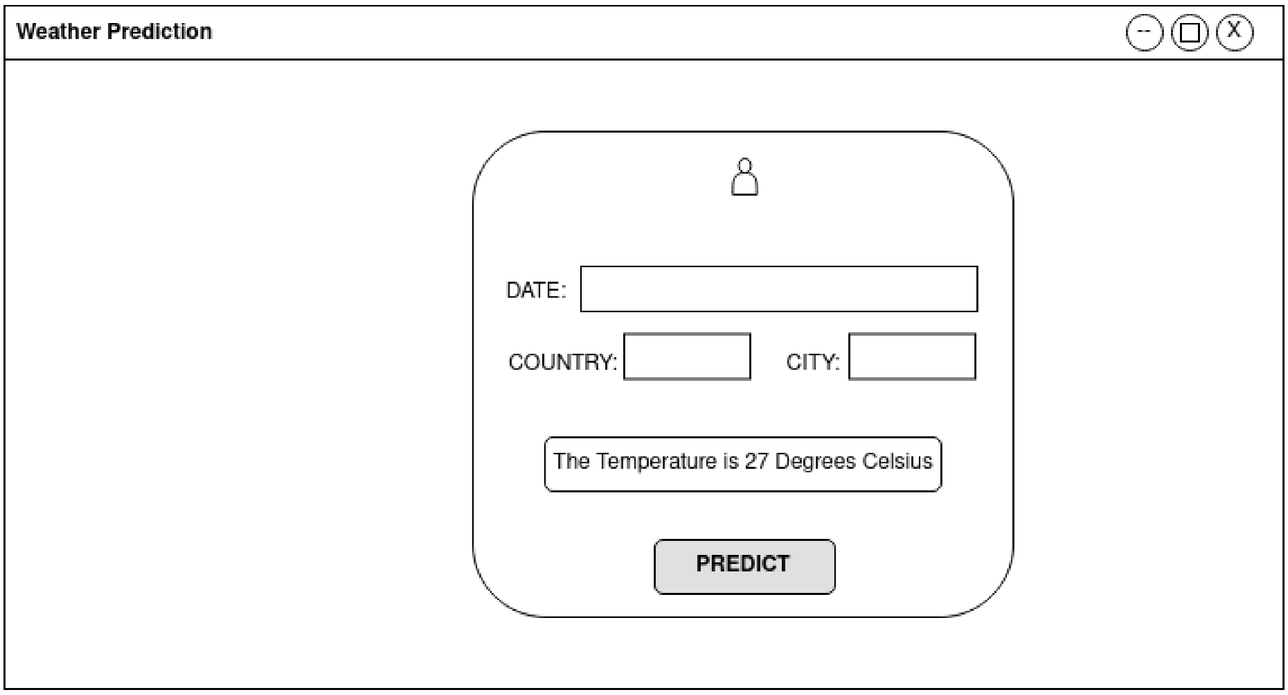
Below is a simple use-case diagram of the application functionalities:



Design Phase

This phase highlights the application's main scenario. Pseudocode and mock-ups are used to explain application flow and logic.

Weather Prediction Information Scenario



1. The system shows the information related to a Weather Prediction input.
2. The User enters all his/her information manually. The date field is a DateTime picker as well as the Country and the City.
3. If the User reloads a page, all inputs are cleared as well as after every Predict.
4. If the User clicks on Predict button
    a. The application preprocesses the data through an API to the Model file and shows the temperature reading within the Field.

UML Diagram

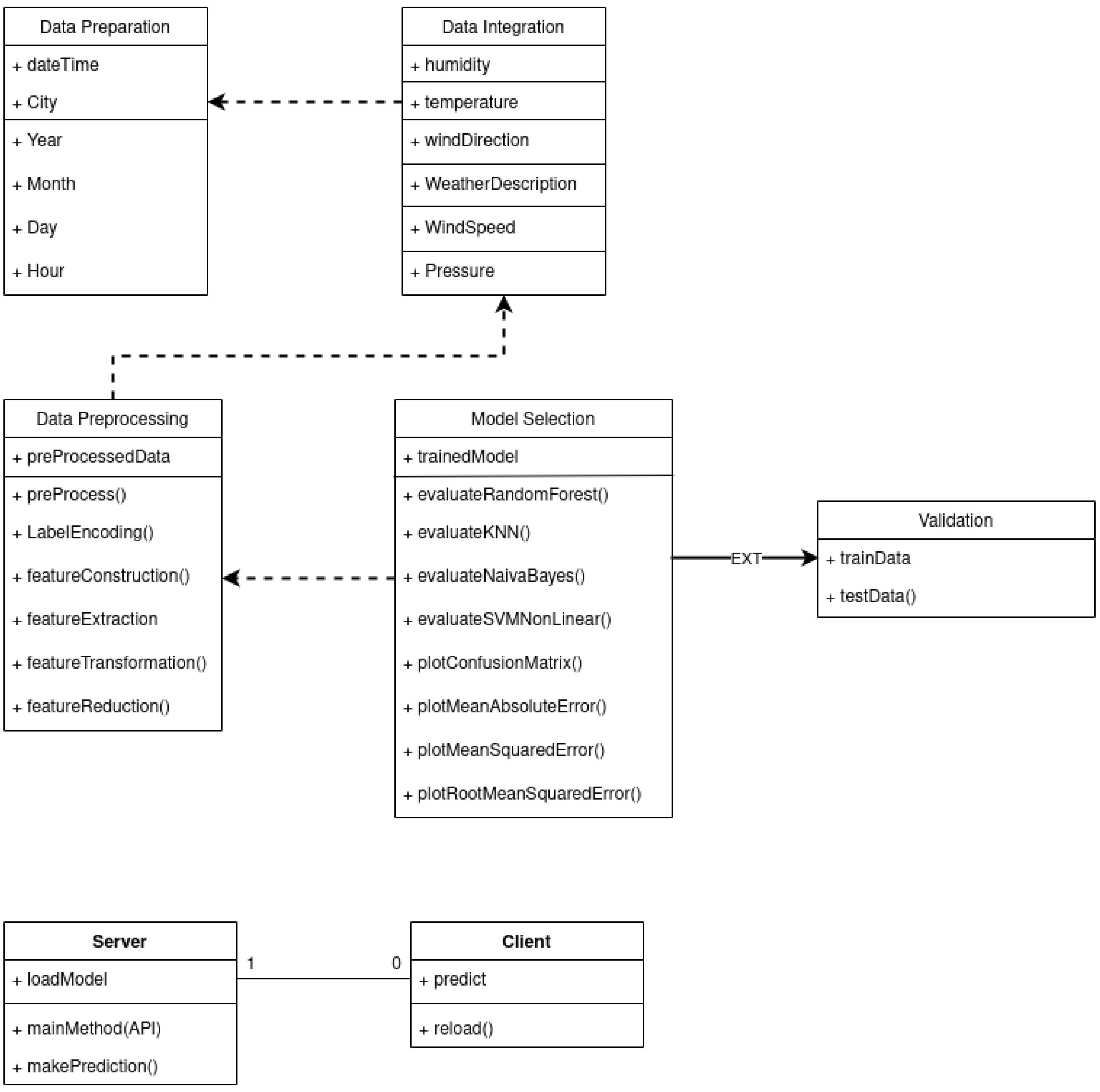
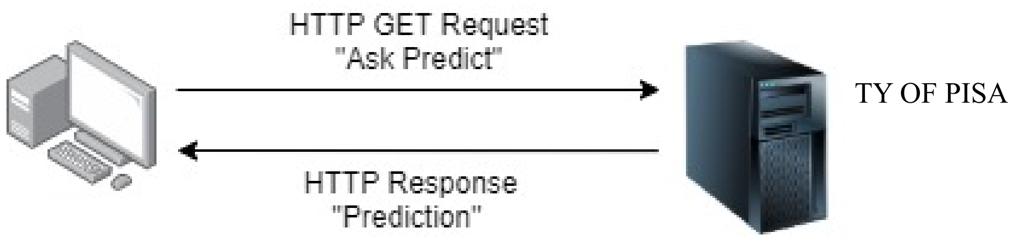


Diagram: UML Class Diagram for Weather Application Flow

Application Architecture

In order to satisfy the non-functional requirement component for the application, specifically, the ease-of-use and portability of the application, a client-server architecture was implemented. The client hits a POST request by making a call to the function and the server responds with the temperature value after consuming the model saved in a pickle file fo



Implementation Phase

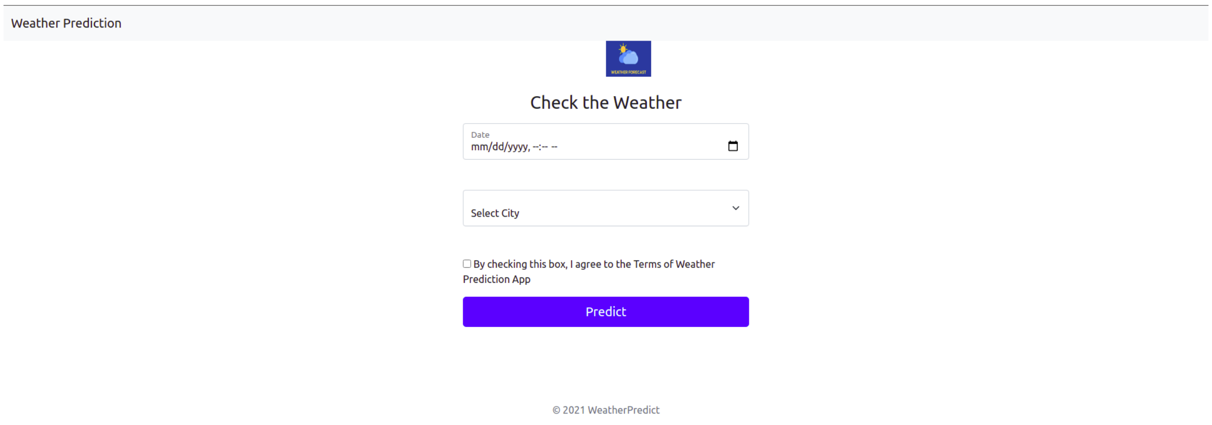
Python was used as the Programming language for the model and data training because of its simplicity and less implementation tasks with dynamic modules and libraries. With the Sklearn library several classification algorithms and implementation techniques like Cross Validation exist which can be achieved with few lines of code. It also allows the use of matplotlib libs for plotting results and visualizations of the outcomes as well as Scipy libraries for analyzing skewness, kurtosis and correlation.
The front-end application is built with React JS and consumed by a Flask API, a micro-framework written in Python. React helps developers build a simple front-end application with components which are reusable with Python Flask, a lightweight service.

The project directory is organized into:

- ➔ **dataset.** The directory where the datasets are loaded in csv format.
- ➔ **scripts.** Contains all the scripts and python notebook files used for data wrangling, manipulation to model building,
- ➔ **weather-prediction.** Contains the front-end react components and node modules installed for the front-end project. This folder also contains an App.js file which renders the components and the pages built and integrated.
- ➔ **api.** This folder contains the server.py file builts with flask which makes a call to the model and returns the response which will be rendered on the interface.

Test Phase

A test phase of the application was done to input values and measure the values against the temperature reading for the City and it proved about 90% accurate with just 1 degree Celsius difference.



Weather Prediction

Check the Weather

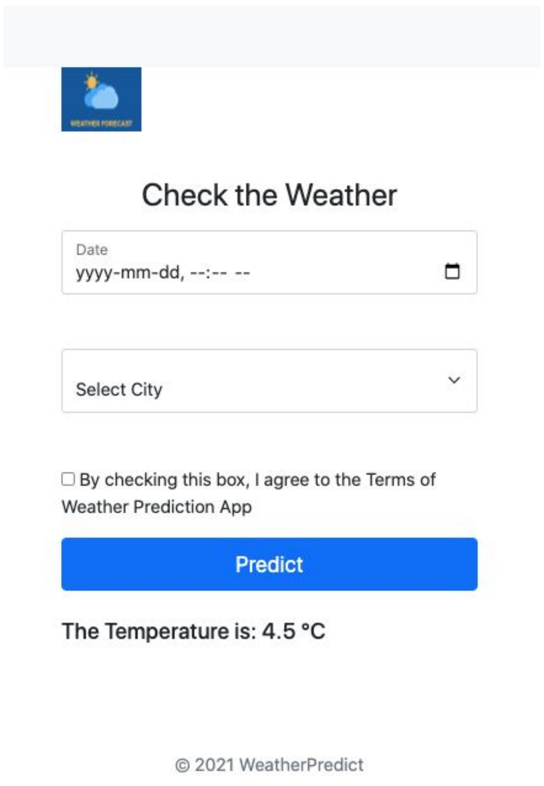
Date
mm/dd/yyyy, --:-- --

Select City

By checking this box, I agree to the Terms of Weather Prediction App

Predict

© 2021 WeatherPredict



Check the Weather

Date
yyyy-mm-dd, --:-- --

Select City

By checking this box, I agree to the Terms of Weather Prediction App

Predict

The Temperature is: 4.5 °C

© 2021 WeatherPredict

User Manual

Preliminary actions

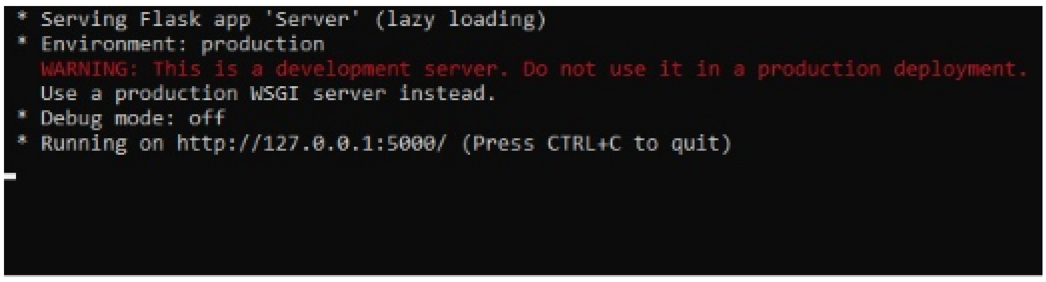
The first thing to do is to install the requirements.txt file for the libraries of the application. Open the command prompt and type the following command:

pip install -r requirements.txt

Afterwards, open the project directory in Terminal or Command Line depending on your operating system. Type the following command:

python app.py

You should see an image like this showing that server is running



```
 * Serving Flask app 'Server' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

How to use the Application

Open a browser and type the command:

http://localhost:5000

The user can choose any date and time from the DateTime picker and select one of the Cities and click on the Predict button.

REFERENCES

https://www.projectpro.io/recipes/find-optimal-parameters-using-randomizedsearchcv-for-regression
https://medium.com/@karanrajwanshi/feature-engineering-the-key-to-predictive-modeling-8f1935b3db4f
https://medium.com/mindorks/what-is-feature-engineering-for-machine-learning-d8ba3158d97a