Summer 15

Weather Data Analytics

CSYE 7374 Big Data Systems & Intelligence Analytics

Group 10: Srinath Sridhar & Shweta Anchan

FINAL PROJECT

Introduction

The project deals with weather data focusing on developing machine learning models predicting the weather condition based on the given weather factors.

Here we look to analyze weather data for UK locations. The dataset contains hourly observations for approximately 150 UK observing stations. These reports are as recorded in real time by the MET Office UK Monitoring System. The parameters in the dataset are based on the instrumentation installed at each site.

Data Gathering

We obtained the data from the following link:

http://data.gov.uk/metoffice-data-archive

This is a Government owned website that provides data for people to understand how the government works in different sectors and the various policies. The data that we are dealing with for this project is an observational data covering several regions of the United Kingdom. The website uploads new data on an hourly basis. Based on our selection, a new CSV file can be downloaded. The data contains information related to weather factors like Wind Direction, Wind Speed, Wind Gust, Visibility, Screen Temperature, Pressure and Pressure Tendency. Also, it has information about the Site Name from which each reading has been taken. It contains the latitude and longitude of the region that the site belongs to. The contents of the file are arranged by a unique site code for easy tracking. The last column of the file contains the type of the weather condition that the reading had led to.

Data Preparation

As the data is being uploaded online on an hourly basis, the selection process in the download page limits people to download one hour of data at a time. So, the main issue here is to get a continuous data for analyzing the contents and working further with them. To describe a little bit more about the issue, we will provide you a simple scenario in the following paragraph.

If we desire to achieve weather information for the last 5 months of this year covering all the sites of UK, we would have to download (5 * 30 * 24) CSV files one by one. Here 5 implies the number of months, 30 implies the number of days [some months have 31 days] and 24 implies the number of hours. Downloading 3600 CSV files manually is a tiresome process and integrating all files into one for creating a continuous data is another tiresome process. So, one of the best methods to opt to for this situation is to create an automation script that will perform these downloads and create a single CSV file with which we will work. We implemented an automation script in Python to make the machine perform the downloads and the integration tasks for us.

There is a package called *mechanize* available in Python that has inbuilt functions to find the forms available on any HTML page and display the types of elements available on them. The form built for the weather data download can be found here http://datagovuk.cloudapp.net/query

How the Automation script works:

Before starting the automation task, you need to create a list of values containing all possible combinations of inputs necessary to feed into to your control block to make the machine download the CSV files. The form has 4 fields namely Query Type, Site Code,

Date of Issue and Time of Issue. Let's consider that you want to get a continuous data for the last 5 months of this year with readings from all the sites. You need to create a list having the Query Type as 'Observational Data', Site Code as 'All Prediction Sites' and Date of Issue as '01/03/2015' and Time of Issue as '0000' as the first element. Then, you need to increment the date and time in your second position and keep continuing until you have the list covering all 5 months of information. This can be achieved by appending contents to a list inside a 'for loop' where the logic to increment time and date will be written. Once you have the list to be passed set up, you find the form and the HTML names given for each element inside the form and create a control block to access all the elements of the form with the found out HTML names. This gives you the power to make the machine download contents for all combinations available in your list. Create a new CSV file before this entire block and include a *file.write* statement at the end of this block to write out the contents of every downloaded CSV file into this file. In this way, you will get a single CSV file containing information obtained from 3600 CSV files.

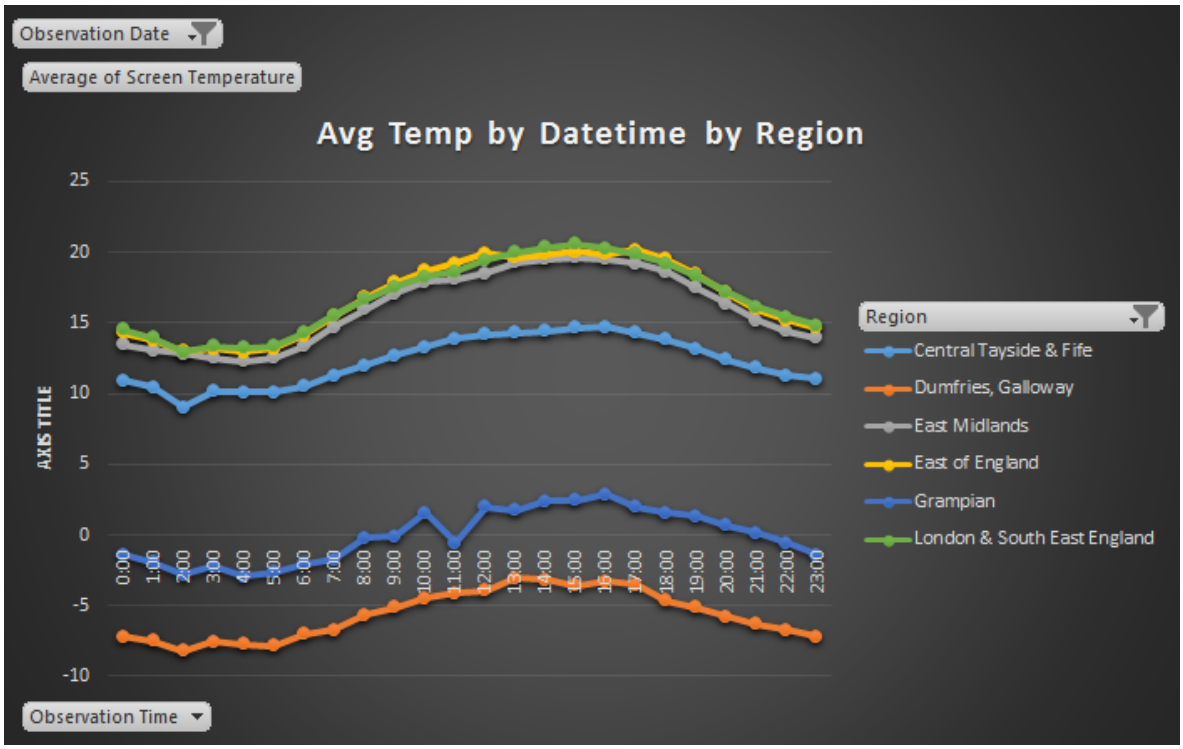
About the tools used:

We use Spark to load and analyze the prepared data. The automation task mentioned above can be performed regularly and have an updated data regularly. This means that we can make the input data dynamic. As our data gets bigger and bigger, it is advisable to work with a tool that does not spill much onto our disk. The reason for choosing Spark over other tools is that Spark does not merge the spilt files into one giant file. We believe we can get a faster execution with Spark rather than other tools like Map Reduce, Python etc.

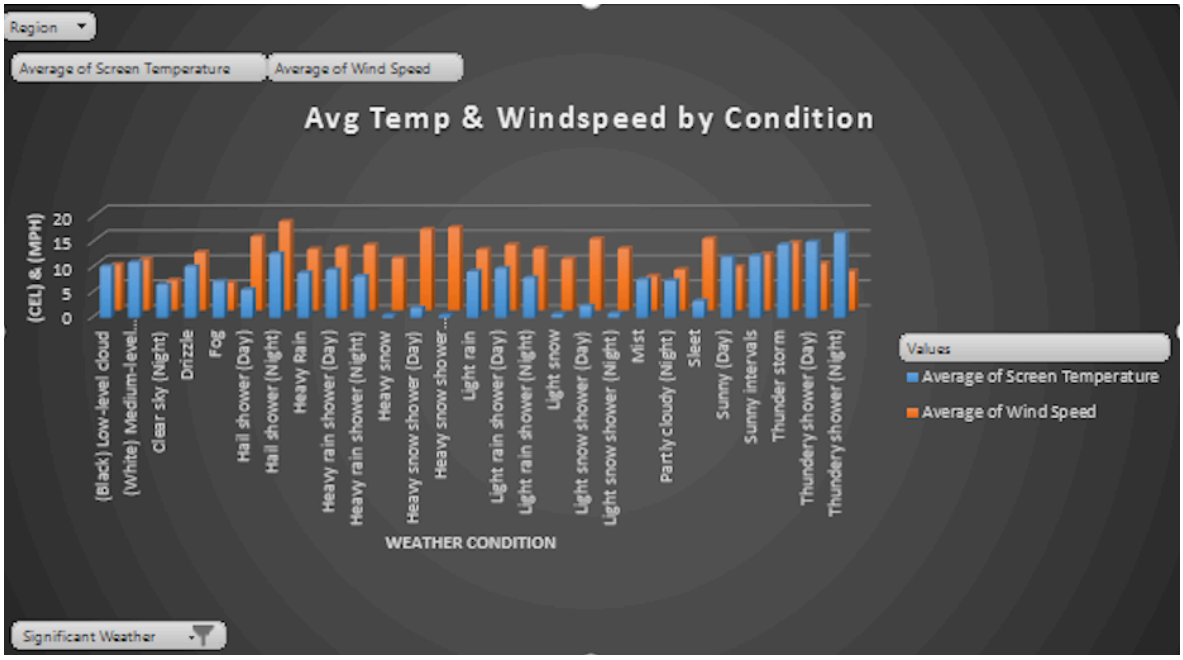
Data Summary

We use tools like Excel and Tableau to create visualizations for better understanding of our data.

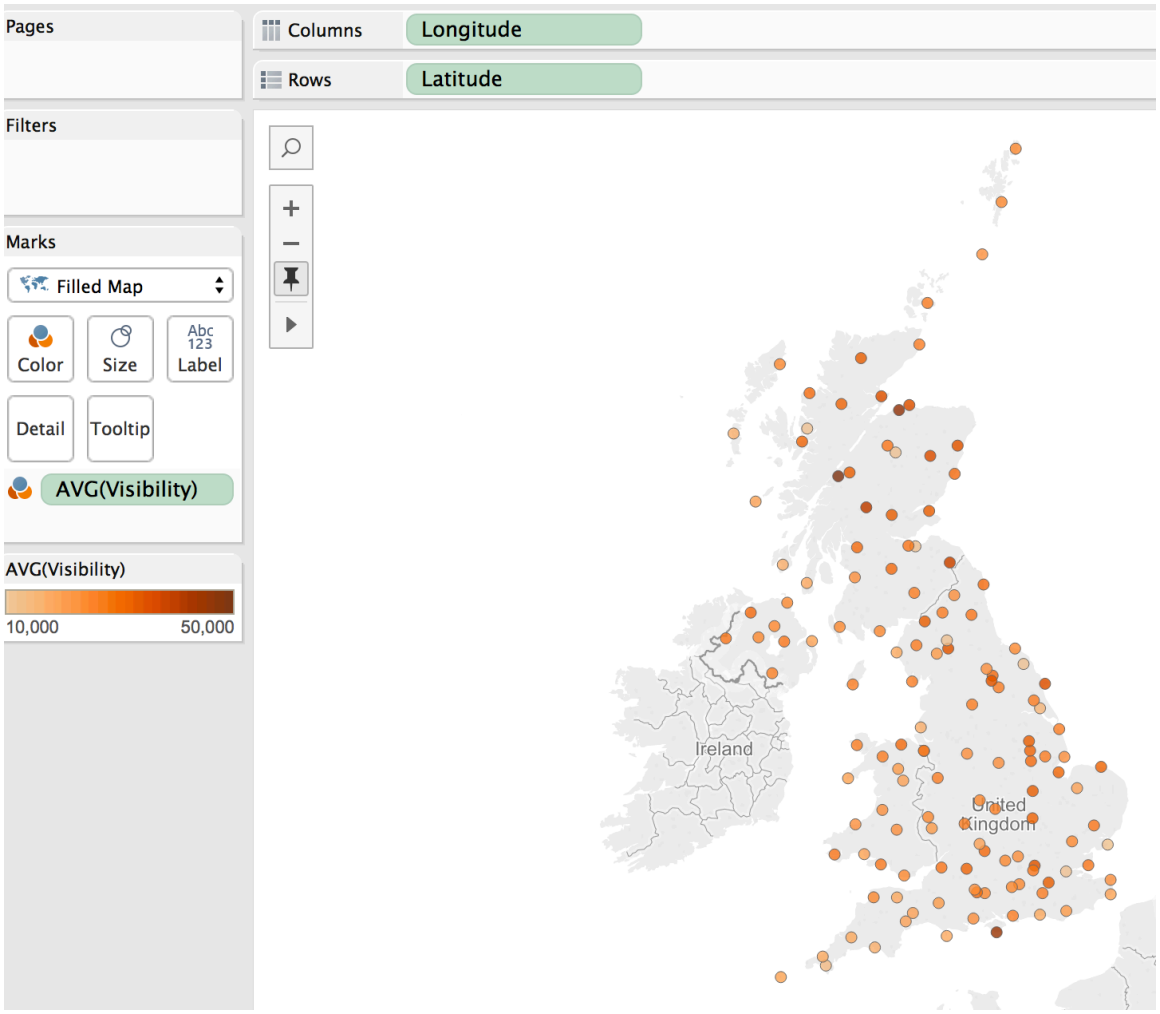
1. The following graph provides us with an understanding of the average temperature for any region over a chosen period of time. This temperature can be filtered for any specific region and by any period.
   From this graph we can analyze that London has a higher average temperature of about 21 Celsius at about 3 p.m. Also we can see that every region experiences its highest temperature at around this time of the afternoon. From the looks of the graph, it looks like Dumfries, Galloway experiences the cold!



2. The next graph shows us the average wind speed and average temperature for a given weather condition. Again, there can be filters for the type of weather or any value.
   We can see that there is an increased Wind Speed when there is a Hail shower or a Snow shower, which is kind of obvious. And also, during heavy snowfall the temperature seems to be least.



3. The next graph speaks for the visibility at each region. Visibility is the measure of the distance at which an object or light can be clearly discerned.
   The graph shows us the visibility of the regions. We plot the regions using the regions Longitude and Latitude. When the dots get more opaque, it means that there is better visibility in those areas.



Data Preprocessing

The biggest challenge in any data science related project is cleansing the data and prepping it to be in a format suitable for performing analytics. This section describes the pre-processing steps taken in this project.

About handling bad data:

The *'Significant Weather'* column has N/A values for some rows and the 'pressure' & 'visibility' column has missing values for some rows. So, the rows with such discrepancies are removed. To do so, we load the data onto a Spark RDD and use a map function that parses the contents line by line to take each line and get the values in those columns and check for N/A or empty values. If there is a match, then we filter out those rows.

The *'Wind Direction'* column represents a 16-point compass covering all the directions available in the compass scale. Also, the column has an unknown value named *'VRB'*. We follow the same procedure to address this value.

About the conversion of categorical data:

Machine learning models in Spark work only with a RDD[LabeledPoint] which in turn expects just float values. So, addressing the categorical columns is a critical task here. Our output column *'Significant Weather'* has 27 unique categorical values. We created a list containing these 27 categories. Then, we iterate over the list item by item using a 'for loop' and assign output value as 1 for the matching condition and 0 otherwise. This will be further explained in the machine learning section.

The type of values available in the *'Wind Direction'* column has been described above. It is a categorical column. A dictionary was created to use a value for each key [unique categories] by ordering the key following the 16-point compass scale and start assigning values from 0 and incrementing by 22.5 for every subsequent key. The number assigned to each key corresponds to the degree they cover in the compass circle.

The *'Observation Time'* column in the data set is in a string format. We create a key-value pair to assign numerical values to them. And, lastly, for the *'Observation Date'* column we take the first letter [0^{th} element] from the date and make it a month. Then, Take the third & fourth letter and made them as date. So, here, the date column will be over written by the month value and the next column will contain the corresponding dates.

About the final preparation:

Once the conversions are set up, a new list of values will be generated and written into the RDD. Now, this RDD will be parsed line by line using a map function to obtain a label and several features. This logic will be written in a separate function and the function returns a LabeledPoint. This RDD[LabeledPoint] will then be given as an input to our machine learning models.

Machine Learning

As the data set contains several categories in the output column, we decided to implement multi-class classification model to train and test the machine. The model is otherwise called a "One Vs Rest" model. In a nutshell, this model addresses all the classes one by one by creating a binary classifier at a time.

About the workflow:

We had already created a list with 27 unique values. These 27 values constitute the entire output column. To apply a multi-class classification logic over this content, we used a 'for loop' to take one element from the list at a time and parse the content of our RDD line by line to obtain the output value for each row. If the output row has the same value as the one looping in the current for loop, then the label will be set as 1. If not, then the label will be set as 0. In this way, we create RDD[LabeledPoint] 27 times and implement the model 27 time and calculate the error rate for each model.

After creating a RDD[LabeledPoint], the classification model is implemented with the training data. The weighted matrix computed by the model is stored in a list. As there are 27 categories, the model gets implemented 27 times and the list will have a [1 *9] weight matrix 27 times. The [1*9] represents the number of inputs we have taken to build the model.

To see how the prediction works with the implemented model, we take a row from the test data and create a [1*9] matrix containing all the features from the selected row and transpose the matrix to get a [9*1] matrix. Then, we perform a matrix multiplication with the transposed one and every [1*9] matrix from the created a list. This matrix multiplication will be done inside a 'for loop' to

perform the multiplication for the entire weight matrix element available inside the list.

The matrix multiplication of [1*9] * [9*1] gives 1 value which implies the value that the model has predicted for each class. After this multiplication, we will be having 27 values and the one with the highest value is the class the model has predicted for the given row.

We tried models like Logistic Regression and SVM With SGD. Also, we reduced the number of output categories to four by providing an abstract name for each category and tried the same models just to compare and contrast the results.

Performing a classification algorithm over a huge number of classes with less than 1 year of data may not train the machine properly. Going by the binary classification logic to address all the output classes will have a higher probability for a ‘class non-match’ condition than a matching condition. For example, if we are implementing a model with the output column having ‘1’ for ‘clear sky (night)’ and ‘0’ for others, the number of 1s being set in a 500,000 records, for this condition, will be very less than the number of 0s. So, when the error rate is calculated, the rate will be very less due to the imbalance in the output class. This does not mean that a proper model has been implemented; it implies that there are too many output classes to classify. So, we tried the same model implementation workflow with a reduced number of output categories. We created an abstract list with 4 values constituting to all the unique categories available in the output column. The result seemed bit better than the first one.

Model Name	Avg Test Error [27 categories]	Avg Test Error [4 cateogires]
LogisticRegressionwithLBFGS	0.032	0.217
SVMwithSGD	0.036	0.317

Screenshots of the prediction:

The test input row we used for prediction is:

test_list = [3321.0,16.0,7.0,30.0,315.0,10.0,45000.0,15.0,1021.0]

The output will give n weighted values where n is equal to the number of unique categories available in the output column.

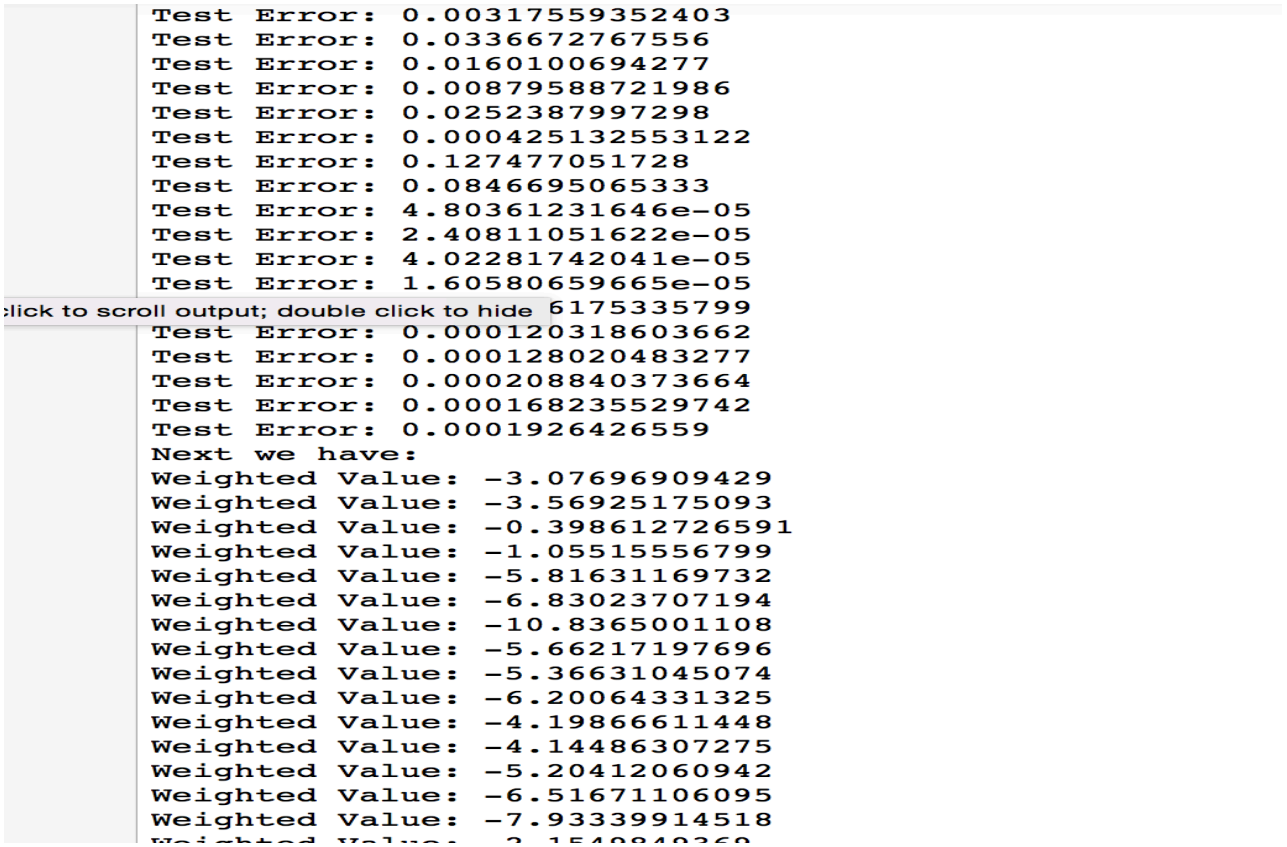
With 4 categories:

```
testError:0.31456651288
testError:0.429365327937
testError:0.125097443563
testError:0.00140172692757
Next we have :
weighted_value:-0.840026592005
weighted_value:0.743047377467
weighted_value:-3.96819890892
weighted_value:-7.19360140586
```

Actual Category: Cloudy Weather

Predicted Category: Cloudy Weather

With 27 categories:



Actual Value: (White) Medium Level Cloud

Predicted Value: Partly Cloudy

The weighted matrix results for SVMWithSGD were too large. So, we believe that the prediction with SVMwithSGD is not very efficient for this data set.

Conclusion

The prediction results were better when the number of output categories were reduced to 4 from 27 categories. The model was able to train the machine properly with such a reduced number of categorical variables. Also, the results were better with the Logistic Regression model than the SVMwithSGD model.