Weather prediction using python

**A Project Report for Industrial Training**

###### **Submitted by**

##### Names of the Team Members

***In the partial fulfillment for the award of the degree of***

##### **B.Tech**

in

Stream

**College Name**

**College Logo**

At

**Ardent Computech Pvt. Ltd.**



**Dec - Jan 2017-2018**



**CERTIFICATE FROM SUPERVISOR**

This is to certify that *[Names of the students]* successfully completed the project titled **"** Weather prediction using python" under my supervision during the period from February to May which is in partial fulfillment of requirements for the award of the Master of Computer Applications and submitted to MCA Department of Future Institute of Engineering and Management, Kolkata.

*Signature of the Supervisor*

**Date:**  **Upasak Pal**

**Project Supervisor**

**Ardent Collaborations**

**Acknowledgement**

The achievement that is associated with the successful completion of any task would be incomplete without mentioning the names of those people whose endless cooperation made it possible. Their constant guidance and encouragement made all our efforts successful.

We take this opportunity to express our deep gratitude towards our project mentor, *[Name of the faculty]* for giving such valuable suggestions, guidance and encouragement during the development of this project work.

Last but not the least we are grateful to all the faculty members of Ardent Computech Pvt. Ltd. for their support.

**Problem Statement**

The goal of the project is to predict the future temperature based off the past three days of weather measurements.

**What is Python**

* Python is a high-level, general-purpose, open source, strictly typed programming language. The language provides constructs intended to enable clear programs on both a small and large scale.
* Python was Created By Guido van Rossum.
* The Python Software Foundation(PSF) is the organization behind Python.

**Python versions**

* First released in 1991.
* Python 2.0 was released on 16 October 2000
* Python 3.0 was released on 3 December 2008
* Current Versions:
  + 3.6.3
  + 2.7.14

**Python features**

Some of the features of python include

* Dynamic
* Object oriented
* Multipurpose
* Strongly typed
* Open Sourced

Python is widely used in many domains

* Web Development
* Data Analysis
* Machine Learning
* Internet Of Things
* GUI Development
* Image processing
* Data visualization
* Game Development

**IDLE**

IDLE is an integrated development environment for Python, which has been bundled with the default implementation of the language

**Anaconda**

Anaconda is a open source Distribution for data science and machine learning using python. It includes hundreds of popular data science packages and the conda package and virtual environment manager for Windows, Linux, and MacOS. Conda makes it quick and easy to install, run, and upgrade complex data science and machine learning environments like scikit-learn, TensorFlow, and SciPy. Anaconda Distribution is the foundation of millions of data science projects as well as Amazon Web Services Machine Learning AMIs and Anaconda for Microsoft on Azure and Windows**.**

**IPython**

IPython is a command shell for interactive computing in multiple programming languages, originally developed for the Python programming language, that offers introspection, rich media, shell syntax, tab completion, and history.

**Packages used**

**Numpy**

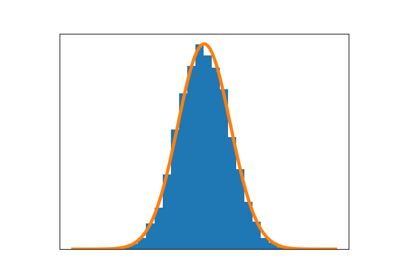
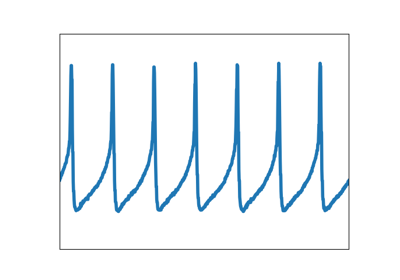
NumPy is the fundamental package for scientific computing with Python. It contains among other things:

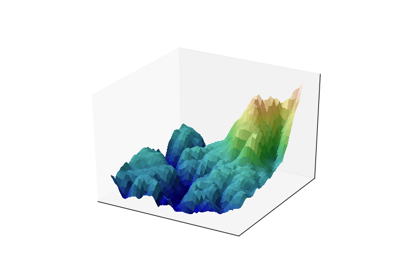
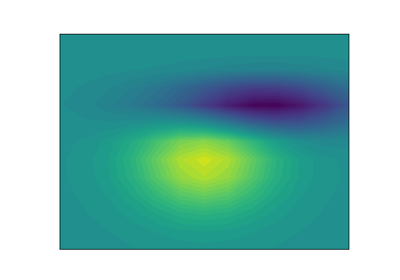
* a powerful N-dimensional array object
* sophisticated (broadcasting) functions
* tools for integrating C/C++ and Fortran code
* useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

**Matplotlib**

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, the jupyter notebook, web application servers, and four graphical user interface toolkits.

[[](https://matplotlib.org/tutorials/introductory/sample_plots.html)](https://matplotlib.org/tutorials/introductory/sample_plots.html)

[[](https://matplotlib.org/tutorials/introductory/sample_plots.html)](https://matplotlib.org/tutorials/introductory/sample_plots.html)

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code.

For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

# scikit-learn

Scikit-learn provides machine learning libraries for python some of the features of Scikit-learn includes:

* Simple and efficient tools for data mining and data analysis
* Accessible to everybody, and reusable in various contexts
* Built on NumPy, SciPy, and matplotlib
* Open source, commercially usable - BSD license

**Pandas**

pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Pandas library is well suited for data manipulation and analysis using python. In particular, it offers data structures and operations for manipulating numerical tables and time series

**Collecting Data From Weather Underground**

Weather Underground is a free tier API web service.

use requests library to interact with the API to pull in weather data since 2015 location Lincoln, Nebraska.

Collected data will be process and aggregated into a format suitable for data analysis, and then cleaned.

Weather Underground

Weather Underground is a company that collects and distributes data on various weather measurements around the globe.

The company provides a swath of API's that are available for both commercial and non-commercial uses.

Account provides an API key to access the web service at a rate of 10 requests per minute and up to a total of 500 requests in a day. The history API provides a summary of various weather measurements for a city and state on a specific day.

Format of the request for the history API resource:

http://[api.wunderground.com/api](http://api.wunderground.com/api)/API\_KEY/history\_YYYYMMDD/q/STATE/CITY.json

* API\_KEY: The API\_KEY that Weather Underground provides with your account
* YYYYMMDD: A string representing the target date of your request
* STATE: The two letter state abbreviation in the United States
* CITY: The name of the city associated with the state you requested

Requests to the API.

| Library | Description of Usage | Source |
| --- | --- | --- |
| [datetime](https://docs.python.org/3/library/datetime.html) | Used to increment our requests by day | Standard Library |
| [time](https://docs.python.org/3/library/time.html) | Used to delay requests to stay under 10 per minute | Standard Library |
| [collections](https://docs.python.org/3/library/collections.html#collections.namedtuple) | Use namedtuples for structured collection of data | Standard Library |
| [pandas](http://pandas.pydata.org/) | Used to process, organize and clean the data | Third Party Library |
| [requests](http://docs.python-requests.org/en/master/) | Used to make networked requests to the API | Third Party Library |
| [matplotlib](https://matplotlib.org/) | Used for graphical analysis | Third Party Library |

from datetime import datetime, timedelta

import time

from collections import namedtuple

import pandas as pd

import requests

import matplotlib.pyplot as plt

constants representing API\_KEY and the BASE\_URL of the API.

Note: you will need to signup for an account with Weather Underground and receive your own API\_KEY.

BASE\_URL is a string with two place holders represented by curly brackets.

The first {} will be filled by the API\_KEY and the second {} will be replaced by a string formatted date. Both values will be interpolated into the BASE\_URL string using the str.format(...) function.

API\_KEY = '898990000090000'

BASE\_URL = "[http://api.wunderground.com/api/{}/history\_{}/q/NE/Lincoln.json](http://api.wunderground.com/api/%7b%7d/history_%7b%7d/q/NE/Lincoln.json)"

Next initialize the target date to the first day of the year in 2015. Then specify the features to parse from the responses returned from the API. The features are simply the keys present in the history -> dailysummary portion of the JSON response. Those features are used to define a namedtuple called DailySummary which I'll use to organize the individual request's data in a list of DailySummary tuples.

target\_date = datetime(2016, 5, 16)

features = ["date", "meantempm", "meandewptm", "meanpressurem", "maxhumidity", "minhumidity", "maxtempm",

"mintempm", "maxdewptm", "mindewptm", "maxpressurem", "minpressurem", "precipm"]

DailySummary = namedtuple("DailySummary", features)

making requests to the API and collecting the successful responses using the function defined below. This function takes the parameters url, api\_key, target\_date and days.

def extract\_weather\_data(url, api\_key, target\_date, days):

records = []

for \_ in range(days):

request = BASE\_URL.format(API\_KEY, target\_date.strftime('%Y%m%d'))

response = requests.get(request)

if response.status\_code == 200:

data = response.json()['history']['dailysummary'][0]

records.append(DailySummary(

date=target\_date,

meantempm=data['meantempm'],

meandewptm=data['meandewptm'],

meanpressurem=data['meanpressurem'],

maxhumidity=data['maxhumidity'],

minhumidity=data['minhumidity'],

maxtempm=data['maxtempm'],

mintempm=data['mintempm'],

maxdewptm=data['maxdewptm'],

mindewptm=data['mindewptm'],

maxpressurem=data['maxpressurem'],

minpressurem=data['minpressurem'],

precipm=data['precipm']))

time.sleep(6)

target\_date += timedelta(days=1)

return records

defining a list called records which will hold the parsed data as DailySummary namedtuples. The for loop is defined so that it iterates over the loop for number of days passed to the function.

Then the request is formatted using the str.format() function to interpolate the API\_KEY and string formatted target\_date object. Once formatted, the request variable is passed to the get() method of the requests object and the response is assigned to a variable called response.

With the response returned make sure the request was successful by evaluating that the HTTP status code is equal to 200. If it is successful then parse the response's body into JSON using the json() method of the returned response object. Chained to the same json() method call I select the indexes of the history and daily summary structures then grab the first item in the dailysummary list and assign that to a variable named data.

The dict-like data structure referenced by the data variable select the desired fields and instantiate a new instance of the DailySummary namedtuple which is appended to the records list.

Finally, each iteration of the loop concludes by calling the sleep method of the time module to pause the loop's execution for six seconds, guaranteeing that no more than 10 requests are made per minute, keeping us within Weather Underground's limits.

Then the target\_date is incremented by 1 day using the timedelta object of the datetime module so the next iteration of the loop retrieves the daily summary for the following day.

The First Batch of Requests

first set of requests for the maximum allotted daily request under the free developer account of 500. the function will take at least an hour depending on network latency. With this we have maxed out our requests for the day, and this is only about half the data we will be working with.come back tomorrow .

records = extract\_weather\_data(BASE\_URL, API\_KEY, target\_date, 500)

Finishing up the Data Retrieval

Ok, now that it is a new day we have a clean slate and up to 500 requests that can be made to the Weather Underground history API. Our batch of 500 requests issued yesterday began on January 1st, 2015 and ended on May 15th, 2016 (assuming you didn't have any failed requests). Once again let us start another batch of 500 requests but, once this last chunk of data is collected we are going to begin formatting it into a Pandas DataFrame and derive potentially useful features.

# if you closed our terminal or Jupyter Notebook, reinitialize your imports and

# variables first and remember to set your target\_date to datetime(2016, 5, 16)

records += extract\_weather\_data(BASE\_URL, API\_KEY, target\_date, 500)

Setting up Pandas DataFrame

Records list of DailySummary named tuples used to build out a Pandas DataFrame. The Pandas DataFrame is a very useful data structure for many programming tasks which are most popularly known for cleaning and processing data to be used in machine learning projects (or experiments).

utilize the Pandas.DataFrame(...) class constructor to instantiate a DataFrame object. The parameters passed to the constructor are records which represent the data for the DataFrame, the features list also used to define the DailySummary namedtuples which will specify the columns of the DataFrame. The set\_index() method is chained to the DataFrame instantiation to specify date as the index.

df = pd.DataFrame(records, columns=features).set\_index('date')

Deriving the Features

we have selected few features while parsing the returned daily summary data to be used in our study. [Machine Learning Applied to Weather Forecasting](https://pdfs.semanticscholar.org/2761/8afb77c5081d942640333528943149a66edd.pdf), they used weather data on the prior two days for the following measurements.

* max temperature
* min temperature
* mean humidity
* mean atmospheric pressure

expanding upon their list of features using the ones listed below, and instead of only using the prior two days we will be going back three days.

* mean temperature
* mean dewpoint
* mean pressure
* max humidity
* min humidity
* max dewpoint
* min dewpoint
* max pressure
* min pressure
* precipitation

So next up is to figure out a way to include these new features as columns in our DataFrame. Make a smaller subset of the current DataFrame to make it easier to work with while developing an algorithm to create these features. I will make a tmp DataFrame consisting of just 10 records and the features meantempm and meandewptm.

tmp = df[['meantempm', 'meandewptm']].head(10)

tmp

| date | meantempm | meandewptm |
| --- | --- | --- |
| 2015-01-01 | -6 | -12 |
| 2015-01-02 | -6 | -9 |
| 2015-01-03 | -4 | -11 |
| 2015-01-04 | -14 | -19 |
| 2015-01-05 | -9 | -14 |
| 2015-01-06 | -10 | -15 |
| 2015-01-07 | -16 | -22 |
| 2015-01-08 | -7 | -12 |
| 2015-01-09 | -11 | -19 |
| 2015-01-10 | -6 | -12 |

Let us break down what we hope to accomplish, and then translate that into code. For each day (row) and for a given feature (column) find the value for that feature N days prior. For each value of N (1-3 in our case) I want to make a new column for that feature representing the Nth prior day's measurement.

# 1 day prior

N = 1

# target measurement of mean temperature

feature = 'meantempm'

# total number of rows

rows = tmp.shape[0]

# a list representing Nth prior measurements of feature

# notice that the front of the list needs to be padded with N

# None values to maintain the constistent rows length for each N

nth\_prior\_measurements = [None]\*N + [tmp[feature][i-N] for i in range(N, rows)]

# make a new column name of feature\_N and add to DataFrame

col\_name = "{}\_{}".format(feature, N)

tmp[col\_name] = nth\_prior\_measurements

tmp

| date | meantempm | meandewptm | meantempm\_1 |
| --- | --- | --- | --- |
| 2015-01-01 | -6 | -12 | None |
| 2015-01-02 | -6 | -9 | -6 |
| 2015-01-03 | -4 | -11 | -6 |
| 2015-01-04 | -14 | -19 | -4 |
| 2015-01-05 | -9 | -14 | -14 |
| 2015-01-06 | -10 | -15 | -9 |
| 2015-01-07 | -16 | -22 | -10 |
| 2015-01-08 | -7 | -12 | -16 |
| 2015-01-09 | -11 | -19 | -7 |
| 2015-01-10 | -6 | -12 | -11 |

We have the basic steps required to make our new features. Now wrap these steps up into a reusable function and put it to work building out all the desired features.

def derive\_nth\_day\_feature(df, feature, N):

rows = df.shape[0]

nth\_prior\_measurements = [None]\*N + [df[feature][i-N] for i in range(N, rows)]

col\_name = "{}\_{}".format(feature, N)

df[col\_name] = nth\_prior\_measurements

Now write a loop to loop over the features in the feature list defined earlier, and for each feature that is not "date" and for N days 1 through 3 we'll call our function to add the derived features we want to evaluate for predicting temperatures.

for feature in features:

if feature != 'date':

for N in range(1, 4):

derive\_nth\_day\_feature(df, feature, N)

And for good measure I will take a look at the columns to make sure that they look as expected.

df.columns

Index(['meantempm', 'meandewptm', 'meanpressurem', 'maxhumidity',

'minhumidity', 'maxtempm', 'mintempm', 'maxdewptm', 'mindewptm',

'maxpressurem', 'minpressurem', 'precipm', 'meantempm\_1', 'meantempm\_2',

'meantempm\_3', 'meandewptm\_1', 'meandewptm\_2', 'meandewptm\_3',

'meanpressurem\_1', 'meanpressurem\_2', 'meanpressurem\_3',

'maxhumidity\_1', 'maxhumidity\_2', 'maxhumidity\_3', 'minhumidity\_1',

'minhumidity\_2', 'minhumidity\_3', 'maxtempm\_1', 'maxtempm\_2',

'maxtempm\_3', 'mintempm\_1', 'mintempm\_2', 'mintempm\_3', 'maxdewptm\_1',

'maxdewptm\_2', 'maxdewptm\_3', 'mindewptm\_1', 'mindewptm\_2',

'mindewptm\_3', 'maxpressurem\_1', 'maxpressurem\_2', 'maxpressurem\_3',

'minpressurem\_1', 'minpressurem\_2', 'minpressurem\_3', 'precipm\_1',

'precipm\_2', 'precipm\_3'],

dtype='object')

The next thing to do is assess the quality of the data and clean it up where necessary.

Data Cleaning

Identify unnecessary data, missing values, consistency of data types, and outliers then making some decisions about how to handle them if they arise.

The goal of the project is to predict the future temperature based off the past three days of weather measurements. With this in mind we only want to keep the min, max, and mean temperatures for each day plus all the new derived variables we added in the last sections.

# make list of original features without meantempm, mintempm, and maxtempm

to\_remove = [feature

for feature in features

if feature not in ['meantempm', 'mintempm', 'maxtempm']]

# make a list of columns to keep

to\_keep = [col for col in df.columns if col not in to\_remove]

# select only the columns in to\_keep and assign to df

df = df[to\_keep]

df.columns

Index(['meantempm', 'maxtempm', 'mintempm', 'meantempm\_1', 'meantempm\_2',

'meantempm\_3', 'meandewptm\_1', 'meandewptm\_2', 'meandewptm\_3',

'meanpressurem\_1', 'meanpressurem\_2', 'meanpressurem\_3',

'maxhumidity\_1', 'maxhumidity\_2', 'maxhumidity\_3', 'minhumidity\_1',

'minhumidity\_2', 'minhumidity\_3', 'maxtempm\_1', 'maxtempm\_2',

'maxtempm\_3', 'mintempm\_1', 'mintempm\_2', 'mintempm\_3', 'maxdewptm\_1',

'maxdewptm\_2', 'maxdewptm\_3', 'mindewptm\_1', 'mindewptm\_2',

'mindewptm\_3', 'maxpressurem\_1', 'maxpressurem\_2', 'maxpressurem\_3',

'minpressurem\_1', 'minpressurem\_2', 'minpressurem\_3', 'precipm\_1',

'precipm\_2', 'precipm\_3'],

dtype='object')

make use of some built in Pandas functions to get a better understanding of the data and potentially identify some areas to focus my energy on. The first function is a DataFrame method called info() which, big surprise... provides information on the DataFrame. Of interest is the "data type" column of the output.

df.info()

<class 'pandas.core.frame.DataFrame'>

DatetimeIndex: 1000 entries, 2015-01-01 to 2017-09-27

Data columns (total 39 columns):

meantempm 1000 non-null object

maxtempm 1000 non-null object

mintempm 1000 non-null object

meantempm\_1 999 non-null object

meantempm\_2 998 non-null object

meantempm\_3 997 non-null object

meandewptm\_1 999 non-null object

meandewptm\_2 998 non-null object

meandewptm\_3 997 non-null object

meanpressurem\_1 999 non-null object

meanpressurem\_2 998 non-null object

meanpressurem\_3 997 non-null object

maxhumidity\_1 999 non-null object

maxhumidity\_2 998 non-null object

maxhumidity\_3 997 non-null object

minhumidity\_1 999 non-null object

minhumidity\_2 998 non-null object

minhumidity\_3 997 non-null object

maxtempm\_1 999 non-null object

maxtempm\_2 998 non-null object

maxtempm\_3 997 non-null object

mintempm\_1 999 non-null object

mintempm\_2 998 non-null object

mintempm\_3 997 non-null object

maxdewptm\_1 999 non-null object

maxdewptm\_2 998 non-null object

maxdewptm\_3 997 non-null object

mindewptm\_1 999 non-null object

mindewptm\_2 998 non-null object

mindewptm\_3 997 non-null object

maxpressurem\_1 999 non-null object

maxpressurem\_2 998 non-null object

maxpressurem\_3 997 non-null object

minpressurem\_1 999 non-null object

minpressurem\_2 998 non-null object

minpressurem\_3 997 non-null object

precipm\_1 999 non-null object

precipm\_2 998 non-null object

precipm\_3 997 non-null object

dtypes: object(39)

memory usage: 312.5+ KB

Notice that the data type of every column is of type "object". We need to convert all of these feature columns to floats for the type of numerical analysis that we hope to perform. To do this I will use the apply() DataFrame method to apply the Pandas to\_numeric method to all values of the DataFrame. The error='coerce' parameter will fill any textual values to [NaNs](https://en.wikipedia.org/wiki/NaN). It is common to find textual values in data from the wild which usually originate from the data collector where data is missing or invalid.

df = df.apply(pd.to\_numeric, errors='coerce')

df.info()

<class 'pandas.core.frame.DataFrame'>

DatetimeIndex: 1000 entries, 2015-01-01 to 2017-09-27

Data columns (total 39 columns):

meantempm 1000 non-null int64

maxtempm 1000 non-null int64

mintempm 1000 non-null int64

meantempm\_1 999 non-null float64

meantempm\_2 998 non-null float64

meantempm\_3 997 non-null float64

meandewptm\_1 999 non-null float64

meandewptm\_2 998 non-null float64

meandewptm\_3 997 non-null float64

meanpressurem\_1 999 non-null float64

meanpressurem\_2 998 non-null float64

meanpressurem\_3 997 non-null float64

maxhumidity\_1 999 non-null float64

maxhumidity\_2 998 non-null float64

maxhumidity\_3 997 non-null float64

minhumidity\_1 999 non-null float64

minhumidity\_2 998 non-null float64

minhumidity\_3 997 non-null float64

maxtempm\_1 999 non-null float64

maxtempm\_2 998 non-null float64

maxtempm\_3 997 non-null float64

mintempm\_1 999 non-null float64

mintempm\_2 998 non-null float64

mintempm\_3 997 non-null float64

maxdewptm\_1 999 non-null float64

maxdewptm\_2 998 non-null float64

maxdewptm\_3 997 non-null float64

mindewptm\_1 999 non-null float64

mindewptm\_2 998 non-null float64

mindewptm\_3 997 non-null float64

maxpressurem\_1 999 non-null float64

maxpressurem\_2 998 non-null float64

maxpressurem\_3 997 non-null float64

minpressurem\_1 999 non-null float64

minpressurem\_2 998 non-null float64

minpressurem\_3 997 non-null float64

precipm\_1 889 non-null float64

precipm\_2 889 non-null float64

precipm\_3 888 non-null float64

dtypes: float64(36), int64(3)

memory usage: 312.5 KB

Now that all of our data has the data type I want I would like to take a look at some summary stats of the features and use the statistical rule of thumb to check for the existence of extreme outliers. The DataFrame method describe() will produce a DataFrame containing the count, mean, standard deviation, min, 25th percentile, 50th percentile (or median), the 75th percentile and, the max value. This can be very useful information to evaluating the distribution of the feature data.

I would like to add to this information by calculating another output column, indicating the existence of outliers. The rule of thumb to identifying an extreme outlier is a value that is less than 3 [interquartile ranges](https://en.wikipedia.org/wiki/Interquartile_range) below the 25th percentile, or 3 interquartile ranges above the 75th percentile. Interquartile range is simply the difference between the 75th percentile and the 25th percentile.

# Call describe on df and transpose it due to the large number of columns

spread = df.describe().T

# precalculate interquartile range for ease of use in next calculation

IQR = spread['75%'] - spread['25%']

# create an outliers column which is either 3 IQRs below the first quartile or

# 3 IQRs above the third quartile

spread['outliers'] = (spread['min']<(spread['25%']-(3\*IQR)))|(spread['max'] > (spread['75%']+3\*IQR))

# just display the features containing extreme outliers

spread.ix[spread.outliers,]

|  | count | mean | std | min | 25% | 50% | 75% | max | outliers |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| maxhumidity\_1 | 999.0 | 88.107107 | 9.273053 | 47.0 | 83.0 | 90.0 | 93.00 | 100.00 | True |
| maxhumidity\_2 | 998.0 | 88.102204 | 9.276407 | 47.0 | 83.0 | 90.0 | 93.00 | 100.00 | True |
| maxhumidity\_3 | 997.0 | 88.093280 | 9.276775 | 47.0 | 83.0 | 90.0 | 93.00 | 100.00 | True |
| maxpressurem\_1 | 999.0 | 1019.924925 | 7.751874 | 993.0 | 1015.0 | 1019.0 | 1024.00 | 1055.00 | True |
| maxpressurem\_2 | 998.0 | 1019.922846 | 7.755482 | 993.0 | 1015.0 | 1019.0 | 1024.00 | 1055.00 | True |
| maxpressurem\_3 | 997.0 | 1019.927783 | 7.757805 | 993.0 | 1015.0 | 1019.0 | 1024.00 | 1055.00 | True |
| minpressurem\_1 | 999.0 | 1012.329329 | 7.882062 | 956.0 | 1008.0 | 1012.0 | 1017.00 | 1035.00 | True |
| minpressurem\_2 | 998.0 | 1012.326653 | 7.885560 | 956.0 | 1008.0 | 1012.0 | 1017.00 | 1035.00 | True |
| minpressurem\_3 | 997.0 | 1012.326981 | 7.889511 | 956.0 | 1008.0 | 1012.0 | 1017.00 | 1035.00 | True |
| precipm\_1 | 889.0 | 2.908211 | 8.874345 | 0.0 | 0.0 | 0.0 | 0.51 | 95.76 | True |
| precipm\_2 | 889.0 | 2.908211 | 8.874345 | 0.0 | 0.0 | 0.0 | 0.51 | 95.76 | True |
| precipm\_3 | 888.0 | 2.888885 | 8.860608 | 0.0 | 0.0 | 0.0 | 0.51 | 95.76 | True |

Assessing the potential impact of outliers is a difficult part of any analytics project. On the one hand, you need to be concerned about the potential for introducing spurious data artifacts that will significantly impact or bias your models. On the other hand, outliers can be extremely meaningful in predicting outcomes that arise under special circumstances. We will discuss each of these outliers containing features and see if we can come to a reasonable conclusion as to how to treat them.

The first set of features all appear to be related to max humidity. Looking at the data I can tell that the outlier for this feature category is due to the apparently very low min value. This indeed looks to be a pretty low value and I think I would like to take a closer look at it, preferably in a graphical way. To do this I will use a histogram.

%matplotlib inline

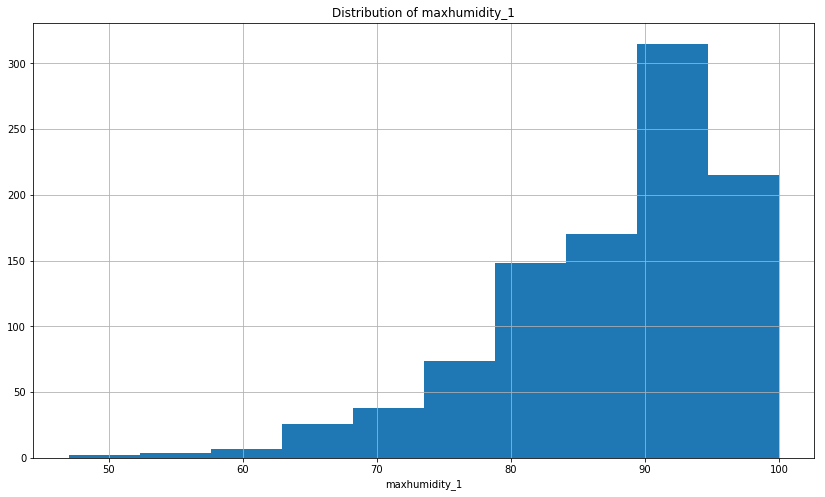
plt.rcParams['figure.figsize'] = [14, 8]

df.maxhumidity\_1.hist()

plt.title('Distribution of maxhumidity\_1')

plt.xlabel('maxhumidity\_1')

plt.show()



Looking at the histogram of the values for maxhumidity the data exhibits quite a bit of negative skew. Keep this in mind when selecting prediction models and evaluating the strength of impact of max humidities. Many of the underlying statistical methods assume that the data is normally distributed.

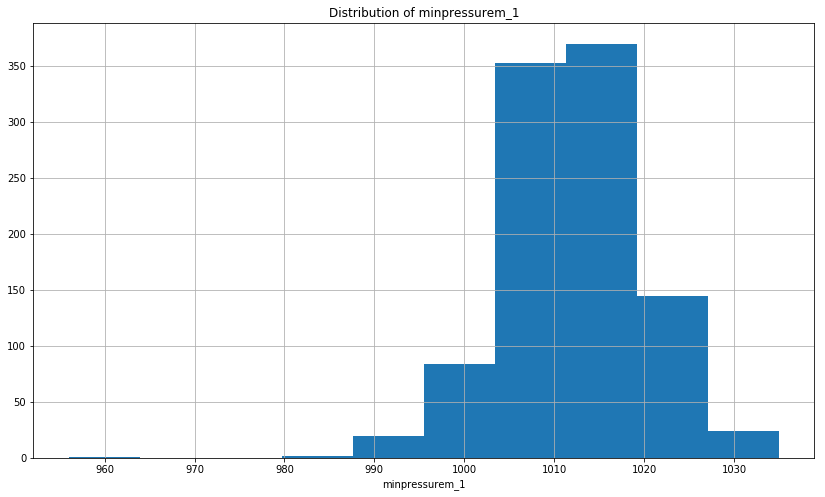
Next look at the minimum pressure feature distribution.

df.minpressurem\_1.hist()

plt.title('Distribution of minpressurem\_1')

plt.xlabel('minpressurem\_1')

plt.show()



The final category of features containing outliers, precipitation, are quite a bit easier to understand. Since the dry days (ie, no precipitation) are much more frequent, it is sensible to see outliers here. To me this is no reason to remove these features.

The last data quality issue to address is that of missing values. Due to the way in which I have built out the DataFrame, the missing values are represented by NaNs. You will probably remember that I have intentionally introduced missing values for the first three days of the data collected by deriving features representing the prior three days of measurements. It is not until the third day in that we can start deriving those features, so clearly I will want to exclude those first three days from the data set.

Look again at the output from the last time I issued the info method. There is a column of output that listed the non-null values for each feature column. Looking at this information you can see that for the most part the features contain relatively few missing (null / NaN) values, mostly just the ones I introduced. However, the precipitation columns appear to be missing a significant part of their data.

Missing data poses a problem because most machine learning methods require complete data sets devoid of any missing data. Aside from the issue that many of the machine learning methods require complete data, if I were to remove all the rows just because the precipitation feature contains missing data then I would be throwing out many other useful feature measurements.

We have a couple of options to deal with this issue of missing data:

1. remove the rows that contain the missing values, but throwing out that much data removes a lot of value from the data
2. fill the missing values with an interpolated value that is a reasonable estimation of the true values.

fill the missing precipitation values with the most common value of zero. majority of values in the precipitation measurements are zero.

# iterate over the precip columns

for precip\_col in ['precipm\_1', 'precipm\_2', 'precipm\_3']:

# create a boolean array of values representing nans

missing\_vals = pd.isnull(df[precip\_col])

df[precip\_col][missing\_vals] = 0

It is quite easy to drop rows from the DataFrame containing NaNs. Call the method dropna() and Pandas will do all the work .

df = df.dropna()

The focus of this will be to describe the processes and steps required to build a rigorous Linear Regression model to predict future mean daily temperature values based off the dataset built in the prior article. To build the Linear Regression model two important Python libraries in the Machine Learning industry: [Scikit-Learn](http://stackabuse.com/the-best-machine-learning-libraries-in-python/#scikitlearn) and [StatsModels](http://www.statsmodels.org/) will be used.

**Importing Dataset**

import pandas as pd

df = pd.read\_csv('end-part2\_df.csv').set\_index('date')

Background on Linear Regression using Ordinary Least Squares

Linear regression aims to apply a set of assumptions primary regarding linear relationships and numerical techniques to predict an outcome (Y, aka the dependent variable) based off of one or more predictors (X's independent variables) with the end goal of establishing a model (mathematical formula) to predict outcomes given only the predictor values with some amount of uncertainty.

The generalized formula for a Linear Regression model is:

ŷ = β0 + β1 \* x1 + β2 \* x2 + ... + β(p-n) x(p-n) + Ε

where:

* ŷ is the predicted outcome variable (dependent variable)
* xj are the predictor variables (independent variables) for j = 1,2,..., p-1 parameters
* β0 is the intercept or the value of ŷ when each xj equals zero
* βj is the change in ŷ based on a one unit change in one of the corresponding xj
* Ε is a random error term associated with the difference between the predicted ŷi value and the actual yi value

That last term in the equation for the Linear Regression is a very important one. The most basic form of building a Linear Regression model relies on an algorithm known as Ordinary Least Squares which finds the combination of βj's values which minimize the Ε term.

**Selecting Features for our Model**

A key assumption required by the linear regression technique is that you have a linear relationship between the dependent variable and each independent variable. One way to assess the linearity between our independent variable, which for now will be the mean temperature, and the other independent variables is to calculate the [Pearson correlation coefficient](https://en.wikipedia.org/wiki/Pearson_correlation_coefficient).

The Pearson correlation coefficient (r) is a measurement of the amount of linear correlation between equal length arrays which outputs a value ranging -1 to 1. Correlation values ranging from 0 to 1 represent increasingly strong positive correlation. By this I mean that two data series are positively correlated when values in one data series increase simultaneously with the values in the other series and, as they both go up in increasingly equal magnitude the Pearson correlation value will approach 1.

Correlation values from 0 to -1 are said to be inversely, or negatively, correlated in that when the values of one series increase the corresponding values in the opposite series decrease but, as changes in magnitude between the series become equal (with opposite direction) the correlation value will approach -1. Pearson correlation values that closely straddle either side of zero are suggestive to have a weak linear relationship, becoming weaker as the value approaches zero.

Opinions vary among statisticians and stats books on clear-cut boundaries for the levels of strength of a correlation coefficient. However, I have found that a generally accepted set of classifications for the strengths of correlation are as follows:

| Correlation Value | Interpretation |
| --- | --- |
| 0.8 - 1.0 | Very Strong |
| 0.6 - 0.8 | Strong |
| 0.4 - 0.6 | Moderate |
| 0.2 - 0.4 | Weak |
| 0.0 - 0.2 | Very Weak |

To assess the correlation in this data I will call the corr() method of the Pandas DataFrame object. Chained to this corr() method call I can then select the column of interest ("meantempm") and again chain another method call sort\_values() on the resulting Pandas Series object. This will output the correlation values from most negatively correlated to the most positively correlated.

df.corr()[['meantempm']].sort\_values('meantempm')

|  | meantempm |
| --- | --- |
| maxpressurem\_1 | -0.519699 |
| maxpressurem\_2 | -0.425666 |
| maxpressurem\_3 | -0.408902 |
| meanpressurem\_1 | -0.365682 |
| meanpressurem\_2 | -0.269896 |
| meanpressurem\_3 | -0.263008 |
| minpressurem\_1 | -0.201003 |
| minhumidity\_1 | -0.148602 |
| minhumidity\_2 | -0.143211 |
| minhumidity\_3 | -0.118564 |
| minpressurem\_2 | -0.104455 |
| minpressurem\_3 | -0.102955 |
| precipm\_2 | 0.084394 |
| precipm\_1 | 0.086617 |
| precipm\_3 | 0.098684 |
| maxhumidity\_1 | 0.132466 |
| maxhumidity\_2 | 0.151358 |
| maxhumidity\_3 | 0.167035 |
| maxdewptm\_3 | 0.829230 |
| maxtempm\_3 | 0.832974 |
| mindewptm\_3 | 0.833546 |
| meandewptm\_3 | 0.834251 |
| mintempm\_3 | 0.836340 |
| maxdewptm\_2 | 0.839893 |
| meandewptm\_2 | 0.848907 |
| mindewptm\_2 | 0.852760 |
| mintempm\_2 | 0.854320 |
| meantempm\_3 | 0.855662 |
| maxtempm\_2 | 0.863906 |
| meantempm\_2 | 0.881221 |
| maxdewptm\_1 | 0.887235 |
| meandewptm\_1 | 0.896681 |
| mindewptm\_1 | 0.899000 |
| mintempm\_1 | 0.905423 |
| maxtempm\_1 | 0.923787 |
| meantempm\_1 | 0.937563 |
| mintempm | 0.973122 |
| maxtempm | 0.976328 |
| meantempm | 1.000000 |

In selecting features to include in this linear regression model, I would like to error on the side of being slightly less permissive in including variables with moderate or lower correlation coefficients. So I will be removing the features that have correlation values less than the absolute value of 0.6. Also, since the "mintempm" and "maxtempm" variables are for the same day as the prediction variable "meantempm", I will be removing those also (i.e. if I already know the min and max temperatures then I already have the answer to my prediction).

With this information, I can now create a new DataFrame that only contains my variables of interest.

predictors = ['meantempm\_1', 'meantempm\_2', 'meantempm\_3',

'mintempm\_1', 'mintempm\_2', 'mintempm\_3',

'meandewptm\_1', 'meandewptm\_2', 'meandewptm\_3',

'maxdewptm\_1', 'maxdewptm\_2', 'maxdewptm\_3',

'mindewptm\_1', 'mindewptm\_2', 'mindewptm\_3',

'maxtempm\_1', 'maxtempm\_2', 'maxtempm\_3']

df2 = df[['meantempm'] + predictors]

Visualizing the Relationships

Because most people, myself included, are much more accustomed to looking at visuals to assess and verify patterns, I will be graphing each of these selected predictors to prove to myself that there is in fact a linear relationship. To do this I will utilize matplotlib's [pyplot](https://matplotlib.org/api/pyplot_api.html) module.

For this plot I would like to have the dependent variable "meantempm" be the consistent y-axis along all of the 18 predictor variables plots. One way to accomplish this is to create a grid of plots. Pandas does come with a useful plotting function called the scatter\_plot(), but I generally only use it when there are only up to about 5 variables because it turns the plot into an N x N matrix (18 x 18 in our case), which becomes difficult to see details in the data. Instead I will create a grid structure with six rows of three columns in order to avoid sacrificing clarity in the graphs.

import matplotlib

import matplotlib.pyplot as plt

import numpy as np

%matplotlib inline

# manually set the parameters of the figure to and appropriate size

plt.rcParams['figure.figsize'] = [16, 22]

# call subplots specifying the grid structure we desire and that

# the y axes should be shared

fig, axes = plt.subplots(nrows=6, ncols=3, sharey=True)

# Since it would be nice to loop through the features in to build this plot

# let us rearrange our data into a 2D array of 6 rows and 3 columns

arr = np.array(predictors).reshape(6, 3)

# use enumerate to loop over the arr 2D array of rows and columns

# and create scatter plots of each meantempm vs each feature

for row, col\_arr in enumerate(arr):

for col, feature in enumerate(col\_arr):

axes[row, col].scatter(df2[feature], df2['meantempm'])

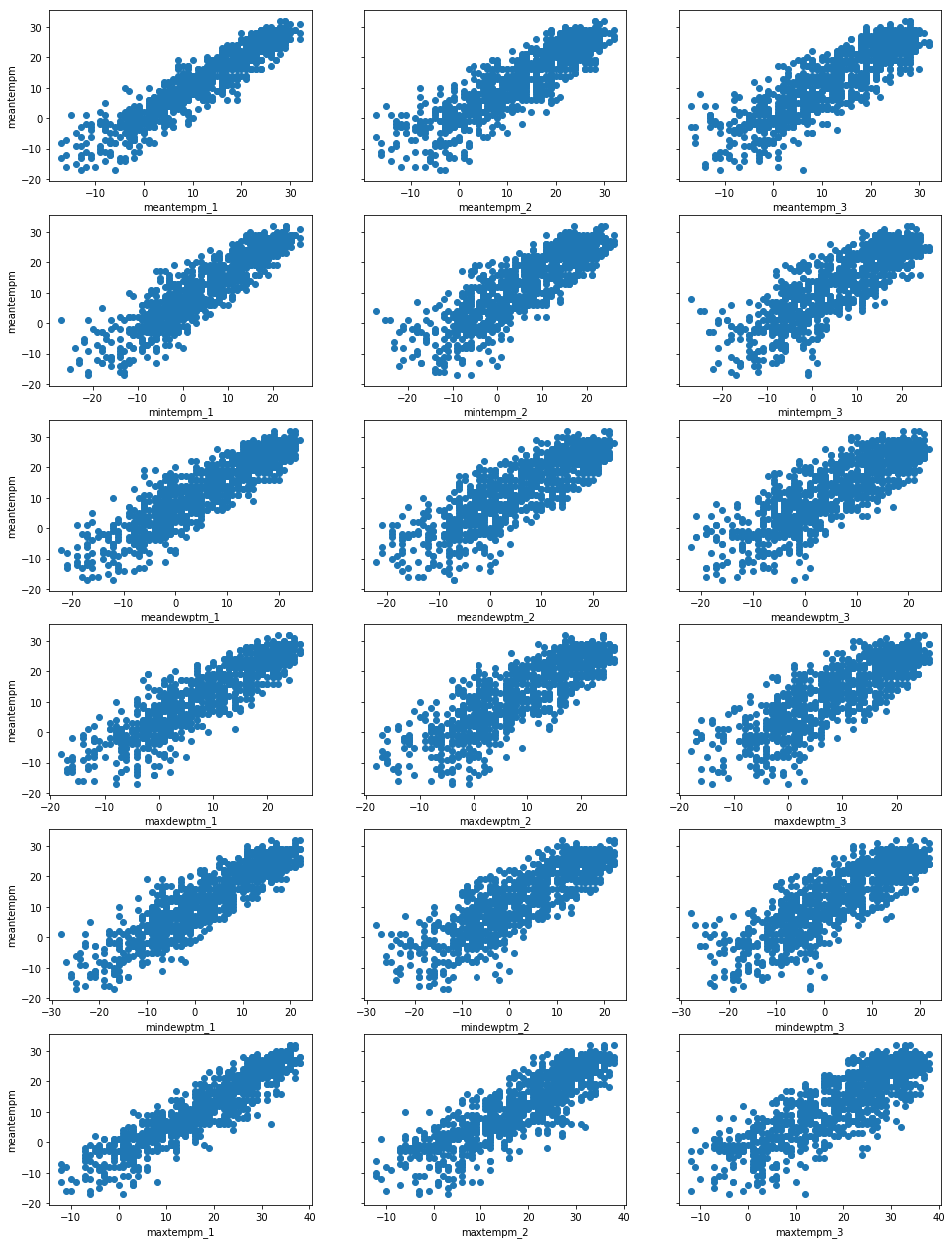
if col == 0:

axes[row, col].set(xlabel=feature, ylabel='meantempm')

else:

axes[row, col].set(xlabel=feature)

plt.show()



From the plots above it is recognizable that all the remaining predictor variables show a good linear relationship with the response variable ("meantempm"). Additionally, it is also worth noting that the relationships all look uniformly randomly distributed. By this I mean there appears to be relatively equal variation in the spread of values devoid of any fanning or cone shape. A uniform random distribution of spread along the points is also another important assumption of Linear Regression using [Ordinary Least Squares](https://en.wikipedia.org/wiki/Ordinary_least_squares) algorithm.

**Using Step-wise Regression to Build a Robust Model**

A robust Linear Regression model should utilize statistical tests for selecting meaningful, statistically significant, predictors to include. To select statistically significant features, I will utilize the Python statsmodels library. However, before I jump into the practical implementation of using the statsmodels library I would like to take a step back and explain some of the theoretical meaning and purpose for taking this approach.

A key aspect of using statistical methods such as Linear Regression in an analytics project are the establishment and testing of hypothesis tests to validate the significance of assumptions made about the data under study. There are numerous hypothesis tests that have been developed to test the robustness of a linear regression model against various assumptions that are made. One such hypothesis test is to evaluate the significance of each of the included predictor variables.

The formal definition of the hypothesis test for the significance of a βj parameters are as follows:

* H0: βj = 0, the null hypothesis states that the predictor has no effect on the outcome variable's value
* Ha: βj ≠ 0, the alternative hypothesis is that the predictor has a significant effect on the outcome variable's value

By using tests of probability to evaluate the likelihood that each βj is significant beyond simple random chance at a selected threshold Α we can be more stringent in selecting the variables to include resulting in a more robust model.

However, in many datasets there can be interactions that occur between variables that can lead to false interpretations of these simple hypothesis tests. To test for the effects of interactions on the significance of any one variable in a linear regression model a technique known as step-wise regression is often applied. Using step-wise regression you add or remove variables from the model and assess the statistical significance of each variable on the resultant model.

We will be using a technique known as backward elimination, where I begin with a fully loaded general model that includes all my variables of interest.

Backward Elimination works as follows:

1. Select a significance level Α for which you test your hypothesis against to determine if a variable should stay in the model
2. Fit the model with all predictor variables
3. Evaluate the p-values of the βj coefficients and for the one with the greatest p-value, if p-value > Α progress to step 4, if not you have your final model
4. Remove the predictor identified in step 3
5. Fit the model again but, this time without the removed variable and cycle back to step 3

So, without further delay let us build this fully loaded generalized model using statsmodels following the above steps.

# import the relevant module

import statsmodels.api as sm

# separate our my predictor variables (X) from my outcome variable y

X = df2[predictors]

y = df2['meantempm']

# Add a constant to the predictor variable set to represent the Bo intercept

X = sm.add\_constant(X)

X.ix[:5, :5]

|  | const | meantempm\_1 | meantempm\_2 | meantempm\_3 | mintempm\_1 |
| --- | --- | --- | --- | --- | --- |
| date |  |  |  |  |  |
| 2015-01-04 | 1.0 | -4.0 | -6.0 | -6.0 | -13.0 |
| 2015-01-05 | 1.0 | -14.0 | -4.0 | -6.0 | -18.0 |
| 2015-01-06 | 1.0 | -9.0 | -14.0 | -4.0 | -14.0 |
| 2015-01-07 | 1.0 | -10.0 | -9.0 | -14.0 | -14.0 |
| 2015-01-08 | 1.0 | -16.0 | -10.0 | -9.0 | -19.0 |

# (1) select a significance value

alpha = 0.05

# (2) Fit the model

model = sm.OLS(y, X).fit()

# (3) evaluate the coefficients' p-values

model.summary()

The summary() call will produce the following data in your Jupyter notebook:

|  |  |  |  |
| --- | --- | --- | --- |
| OLS Regression Results | | | |
| Dep. Variable: | meantempm | R-squared: | 0.895 |
| Model: | OLS | Adj. R-squared: | 0.893 |
| Method: | Least Squares | F-statistic: | 462.7 |
| Date: | Thu, 16 Nov 2017 | Prob (F-statistic): | 0.00 |
| Time: | 20:55:25 | Log-Likelihood: | -2679.2 |
| No. Observations: | 997 | AIC: | 5396. |
| Df Residuals: | 978 | BIC: | 5490. |
| Df Model: | 18 |  |  |
| Covariance Type: | nonrobust |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | coef | std err | t | P>|t| | [0.025 | 0.975] |
| const | 1.0769 | 0.526 | 2.049 | 0.041 | 0.046 | 2.108 |
| meantempm\_1 | 0.1047 | 0.287 | 0.364 | 0.716 | -0.459 | 0.669 |
| meantempm\_2 | 0.3512 | 0.287 | 1.225 | 0.221 | -0.211 | 0.914 |
| meantempm\_3 | -0.1084 | 0.286 | -0.379 | 0.705 | -0.669 | 0.453 |
| mintempm\_1 | 0.0805 | 0.149 | 0.539 | 0.590 | -0.213 | 0.373 |
| mintempm\_2 | -0.2371 | 0.149 | -1.587 | 0.113 | -0.530 | 0.056 |
| mintempm\_3 | 0.1521 | 0.148 | 1.028 | 0.304 | -0.138 | 0.443 |
| meandewptm\_1 | -0.0418 | 0.138 | -0.304 | 0.761 | -0.312 | 0.228 |
| meandewptm\_2 | -0.0121 | 0.138 | -0.088 | 0.930 | -0.282 | 0.258 |
| meandewptm\_3 | -0.0060 | 0.137 | -0.044 | 0.965 | -0.275 | 0.263 |
| maxdewptm\_1 | -0.1592 | 0.091 | -1.756 | 0.079 | -0.337 | 0.019 |
| maxdewptm\_2 | -0.0113 | 0.091 | -0.125 | 0.900 | -0.189 | 0.166 |
| maxdewptm\_3 | 0.1326 | 0.089 | 1.492 | 0.136 | -0.042 | 0.307 |
| mindewptm\_1 | 0.3638 | 0.084 | 4.346 | 0.000 | 0.200 | 0.528 |
| mindewptm\_2 | -0.0119 | 0.088 | -0.136 | 0.892 | -0.184 | 0.160 |
| mindewptm\_3 | -0.0239 | 0.086 | -0.279 | 0.780 | -0.192 | 0.144 |
| maxtempm\_1 | 0.5042 | 0.147 | 3.438 | 0.001 | 0.216 | 0.792 |
| maxtempm\_2 | -0.2154 | 0.147 | -1.464 | 0.143 | -0.504 | 0.073 |
| maxtempm\_3 | 0.0809 | 0.146 | 0.555 | 0.579 | -0.205 | 0.367 |

|  |  |  |  |
| --- | --- | --- | --- |
| Omnibus: | 13.252 | Durbin-Watson: | 2.015 |
| Prob(Omnibus): | 0.001 | Jarque-Bera (JB): | 17.097 |
| Skew: | -0.163 | Prob(JB): | 0.000194 |
| Kurtosis: | 3.552 | Cond. No. | 291. |

Ok, I recognize that the call to summary() just barfed out a whole lot of information onto the screen. Do not get overwhelmed! We are only going to focus on about 2-3 values in this article:

1. P>|t| - this is the p-value I mentioned above that I will be using to evaluate the hypothesis test. This is the value we are going to use to determine whether to eliminate a variable in this step-wise backward elimination technique.
2. R-squared - a measure that states how much of the overall variance in the outcome our model can explain
3. Adj. R-squared - the same as R-squared but, for multiple linear regression this value has a penalty applied to it based off the number of variables being included to explain the level of overfitting.

# (3) cont. - Identify the predictor with the greatest p-value and assess if its > our selected alpha.

# based off the table it is clear that meandewptm\_3 has the greatest p-value and that it is

# greater than our alpha of 0.05

# (4) - Use pandas drop function to remove this column from X

X = X.drop('meandewptm\_3', axis=1)

# (5) Fit the model

model = sm.OLS(y, X).fit()

model.summary()

|  |  |  |  |
| --- | --- | --- | --- |
| OLS Regression Results | | | |
| Dep. Variable: | meantempm | R-squared: | 0.895 |
| Model: | OLS | Adj. R-squared: | 0.893 |
| Method: | Least Squares | F-statistic: | 490.4 |
| Date: | Thu, 16 Nov 2017 | Prob (F-statistic): | 0.00 |
| Time: | 20:55:41 | Log-Likelihood: | -2679.2 |
| No. Observations: | 997 | AIC: | 5394. |
| Df Residuals: | 979 | BIC: | 5483. |
| Df Model: | 17 |  |  |
| Covariance Type: | nonrobust |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | coef | std err | t | P>|t| | [0.025 | 0.975] |
| const | 1.0771 | 0.525 | 2.051 | 0.041 | 0.046 | 2.108 |
| meantempm\_1 | 0.1040 | 0.287 | 0.363 | 0.717 | -0.459 | 0.667 |
| meantempm\_2 | 0.3513 | 0.286 | 1.226 | 0.220 | -0.211 | 0.913 |
| meantempm\_3 | -0.1082 | 0.286 | -0.379 | 0.705 | -0.669 | 0.452 |
| mintempm\_1 | 0.0809 | 0.149 | 0.543 | 0.587 | -0.211 | 0.373 |
| mintempm\_2 | -0.2371 | 0.149 | -1.588 | 0.113 | -0.530 | 0.056 |
| mintempm\_3 | 0.1520 | 0.148 | 1.028 | 0.304 | -0.138 | 0.442 |
| meandewptm\_1 | -0.0419 | 0.137 | -0.305 | 0.761 | -0.312 | 0.228 |
| meandewptm\_2 | -0.0121 | 0.138 | -0.088 | 0.930 | -0.282 | 0.258 |
| maxdewptm\_1 | -0.1592 | 0.091 | -1.757 | 0.079 | -0.337 | 0.019 |
| maxdewptm\_2 | -0.0115 | 0.090 | -0.127 | 0.899 | -0.189 | 0.166 |
| maxdewptm\_3 | 0.1293 | 0.048 | 2.705 | 0.007 | 0.036 | 0.223 |
| mindewptm\_1 | 0.3638 | 0.084 | 4.349 | 0.000 | 0.200 | 0.528 |
| mindewptm\_2 | -0.0119 | 0.088 | -0.135 | 0.892 | -0.184 | 0.160 |
| mindewptm\_3 | -0.0266 | 0.058 | -0.456 | 0.648 | -0.141 | 0.088 |
| maxtempm\_1 | 0.5046 | 0.146 | 3.448 | 0.001 | 0.217 | 0.792 |
| maxtempm\_2 | -0.2154 | 0.147 | -1.465 | 0.143 | -0.504 | 0.073 |
| maxtempm\_3 | 0.0809 | 0.146 | 0.556 | 0.579 | -0.205 | 0.367 |

|  |  |  |  |
| --- | --- | --- | --- |
| Omnibus: | 13.254 | Durbin-Watson: | 2.015 |
| Prob(Omnibus): | 0.001 | Jarque-Bera (JB): | 17.105 |
| Skew: | -0.163 | Prob(JB): | 0.000193 |
| Kurtosis: | 3.553 | Cond. No. | 286. |

In respect of your reading time and in an attempt to keep the article to a reasonable length I am going to omit the remaining elimination cycles required to build each new model, evaluate p-values and remove the least significant value. Instead I will jump right to the last cycle and provide you with the final model. Afterall, the main goal here was to describe the process and the reasoning behind it.

Below you will find the output from the final model I converged on after applying the backwards elimination technique. You can see from the output that all the remaining predictors have a p-values significantly below our Α of 0.05. Another thing worthy of some attention are the R-squared values in the final output. Two things to note here are (1) the R-squared and Adj. R-squared values are both equal which suggests there is minimal risk that our model is being over fitted by excessive variables and (2) the value of 0.894 is interpreted such that our final model explains about 90% of the observed variation in the outcome variable, the "meantempm".

model = sm.OLS(y, X).fit()

model.summary()

|  |  |  |  |
| --- | --- | --- | --- |
| OLS Regression Results | | | |
| Dep. Variable: | meantempm | R-squared: | 0.894 |
| Model: | OLS | Adj. R-squared: | 0.894 |
| Method: | Least Squares | F-statistic: | 1196. |
| Date: | Thu, 16 Nov 2017 | Prob (F-statistic): | 0.00 |
| Time: | 20:55:47 | Log-Likelihood: | -2681.7 |
| No. Observations: | 997 | AIC: | 5379. |
| Df Residuals: | 989 | BIC: | 5419. |
| Df Model: | 7 |  |  |
| Covariance Type: | nonrobust |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | coef | std err | t | P>|t| | [0.025 | 0.975] |
| const | 1.1534 | 0.411 | 2.804 | 0.005 | 0.346 | 1.961 |
| mintempm\_1 | 0.1310 | 0.053 | 2.458 | 0.014 | 0.026 | 0.236 |
| mintempm\_2 | -0.0964 | 0.037 | -2.620 | 0.009 | -0.169 | -0.024 |
| mintempm\_3 | 0.0886 | 0.041 | 2.183 | 0.029 | 0.009 | 0.168 |
| maxdewptm\_1 | -0.1939 | 0.047 | -4.117 | 0.000 | -0.286 | -0.101 |
| maxdewptm\_3 | 0.1269 | 0.040 | 3.191 | 0.001 | 0.049 | 0.205 |
| mindewptm\_1 | 0.3352 | 0.051 | 6.605 | 0.000 | 0.236 | 0.435 |
| maxtempm\_1 | 0.5506 | 0.024 | 22.507 | 0.000 | 0.503 | 0.599 |

|  |  |  |  |
| --- | --- | --- | --- |
| Omnibus: | 13.123 | Durbin-Watson: | 1.969 |
| Prob(Omnibus): | 0.001 | Jarque-Bera (JB): | 16.871 |
| Skew: | -0.163 | Prob(JB): | 0.000217 |
| Kurtosis: | 3.548 | Cond. No. | 134. |

Using SciKit-Learn's LinearRegression Module to Predict the Weather

Now that we have gone through the steps to select statistically meaningful predictors (features), we can use [SciKit-Learn](http://scikit-learn.org/) to create a prediction model and test its ability to predict the mean temperature. SciKit-Learn is a very well established machine learning library that is widely used in both industry and academia. One thing that is very impressive about SciKit-Learn is that it maintains a very consistent API of "fit", "predict", and "test" across many numerical techniques and algorithms which makes using it very simple. In addition to this consistent API design, SciKit-Learn also comes with several useful tools for processing data common to many machine learning projects.

We will start by using SciKit-Learn to split our dataset into a testing and training sets by importing the train\_test\_split() function from sklearn.model\_selection module. I will split the training and testing datasets into 80% training and 20% testing and assign a random\_state of 12 to ensure you will get the same random selection of data as I do. This random\_state parameter is very useful for reproducibility of results.

from sklearn.model\_selection import train\_test\_split

# first remove the const column because unlike statsmodels, SciKit-Learn will add that in for us

X = X.drop('const', axis=1)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=12)

The next action to take is to build the regression model using the training dataset. To do this I will import and use the LinearRegression class from the sklearn.linear\_model module. As mentioned previously, scikit-learn scores major usability bonus points by implementing a common fit() and predict() API across its numerous numerical techniques which makes using the library very user friendly.

from sklearn.linear\_model import LinearRegression

# instantiate the regressor class

regressor = LinearRegression()

# fit the build the model by fitting the regressor to the training data

regressor.fit(X\_train, y\_train)

# make a prediction set using the test set

prediction = regressor.predict(X\_test)

# Evaluate the prediction accuracy of the model

from sklearn.metrics import mean\_absolute\_error, median\_absolute\_error

print("The Explained Variance: %.2f" % regressor.score(X\_test, y\_test))

print("The Mean Absolute Error: %.2f degrees celsius" % mean\_absolute\_error(y\_test, prediction))

print("The Median Absolute Error: %.2f degrees celsius" % median\_absolute\_error(y\_test, prediction))

The Explained Variance: 0.90

The Mean Absolute Error: 2.69 degrees celsius

The Median Absolute Error: 2.17 degrees celsius

Conclusion

As you can see in the few lines of code above using scikit-learn to build a Linear Regression prediction model is quite simple. This is truly where the library shines in its ability to easily fit a model and make predictions about an outcome of interest.

To gain an interpretative understanding of the models validity use the regressor model's score() function to determine that the model is able to explain about 90% of the variance observed in the outcome variable, mean temperature.

Additionally, use the mean\_absolute\_error() and median\_absolute\_error() of the sklearn.metrics module to determine that on average the predicted value is about 3 degrees Celsius off and half of the time it is off by about 2 degrees Celsius.

**Sources**

<https://www.python.org/>

<https://anaconda.org/anaconda/python>

<http://www.numpy.org/>

<https://matplotlib.org/>

<http://scikit-learn.org/>

<https://pandas.pydata.org/>

<https://pandas.pydata.org/>

<https://ipython.org/>