Linear Regression

Exercise 1: Plotting Data with a Moving Average

Let's use this exercise to load, plot, and interrogate the data source:

1. Import the **`numpy`**, **`pandas`**, and **`matplotlib`** packages:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```
2. Use the pandas **`read_csv`** function to load the CSV file containing the **`synth_temp.csv`** dataset, and then display the first five lines of data:
```
df =
pd.read_csv('../Datasets/synth_temp.csv'
)
df.head()
```

The output will be as follows:

	Region	Year	RgnAvTemp
0	A	1841	12.557395
1	B	1841	13.267048
2	E	1841	12.217463
3	F	1841	13.189420
4	A	1842	13.462887

Figure 1: The first five rows

3. For our purposes, we don't want to use all this data, but let's look at how many points there are per year. Create a **`print`** statement to output the number of points for the years 1841, 1902, and 2010, and make a simple plot of the number of points per year:

```
# take a quick look at the number of
data points per year
print('There are ' +
str(len(df.loc[df['Year'] == 1841])) \
      + ' points in 1841\n' + 'and ' \
      + str(len(df.loc[df['Year'] ==
2010])) \
      + ' points in 2010\n' + 'and ' \
      + str(len(df.loc[df['Year'] ==
1902])) \
      + ' points in 1902')
# seeing there are different numbers of
points, let's do a quick chart
fig, ax = plt.subplots()
ax.plot(df['Year'].unique(),
[len(df.loc[df['Year'] == i]) \
        for i in df['Year'].unique()])
plt.show()
```

The output will be as follows:

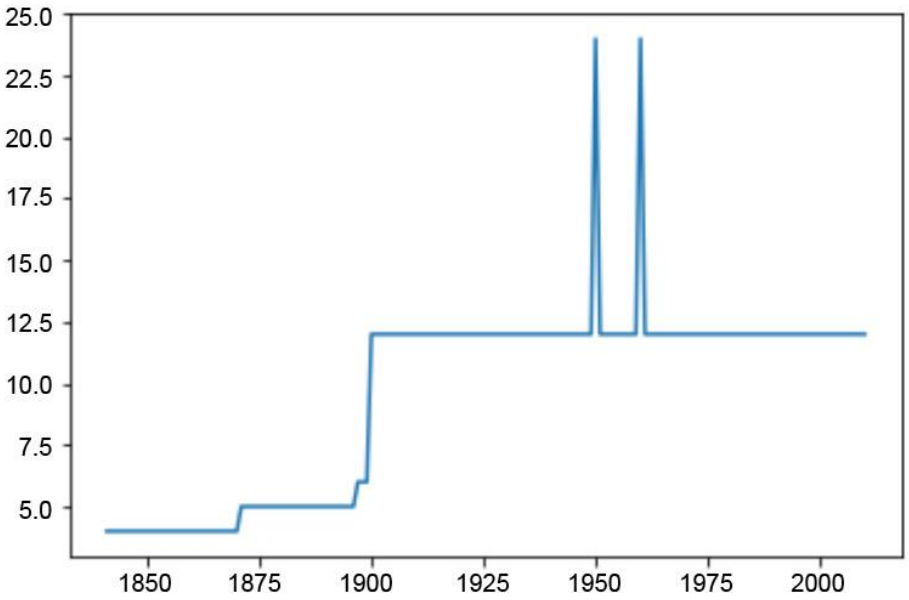

Figure 2: Different number of points per year

We see varying numbers of points per year.

Also note that we don't have the information on exactly when in each year the various points were measured. If that were important, we would want to ask the appropriate business stakeholder if the information could be obtained.

4. Let's slice the DataFrame to remove all rows through 1901, as we can see that there is much less data in those years:

```
# slice 1902 and forward
df = df.loc[df.Year > 1901]
df.head()
```

The output will be as follows:

	Region	Year	RgnAvTemp
292	A	1902	17.021583
293	B	1902	17.590253
294	C	1902	17.493082
295	D	1902	18.706166
296	E	1902	17.390903

Figure 3: Subset of data from 1902 onward

5. Make a quick plot to visualize the data:

```
# quick plot to understand what we have so far
fig, ax = plt.subplots()
ax.scatter(df.Year, df.RgnAvTemp)
plt.show()
```

The output will be as follows:

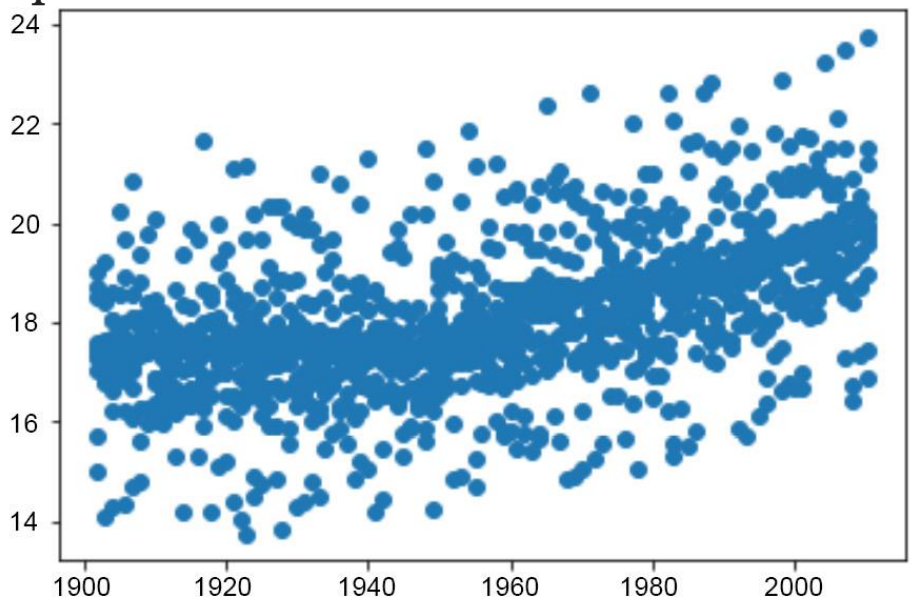

Figure 4: Basic visualization of raw data after filtering dates

6. We can see that there is quite a range for each year. Group the data by year and use the `agg` method of the DataFrame to create annual averages. This works around the issue

that we have multiple points at unknown dates in each year, but uses all the data:

```
# roll up by year
df_group_year =
(df.groupby('Year').agg('mean')\
                .rename(columns =
{'RgnAvTemp' : 'AvgTemp'}))
print(df_group_year.head())
print(df_group_year.tail())
```

The output will be as follows:

```
          AvgTemp
Year
1902  17.385044
1903  17.222163
1904  17.217215
1905  17.817502
1906  17.386445
          AvgTemp
Year
2006  19.904999
2007  19.820224
2008  19.245558
2009  19.537290
2010  19.919115
```

Figure 5: Yearly average data

As before, perform a quick visualization, as follows:

```
# visualize result of averaging over
each year
fig, ax = plt.subplots()
ax.scatter(df_group_year.index,
df_group_year['AvgTemp'])
plt.show()
```

The data will now appear as follows:

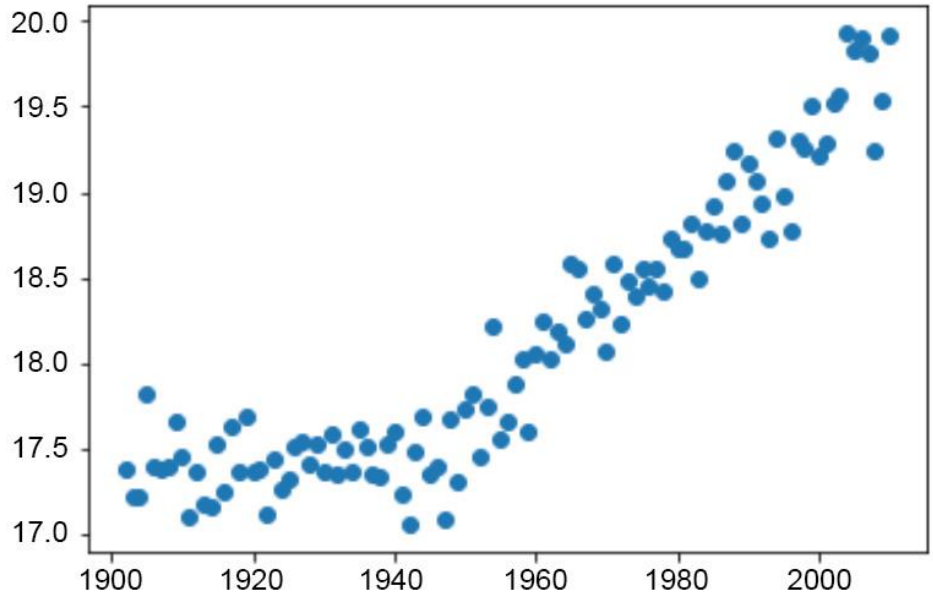

Figure 6: Yearly average data

7. Given that the data is still noisy, a moving average filter can provide a useful indicator of the overall trend. A moving average filter simply computes the average over the last *N* values and assigns this average to the *N*th sample. Compute the values for a moving average signal for the temperature measurements using a window of 10 years:

```
window = 10
smoothed_df = \
pd.DataFrame(df_group_year.AvgTemp.rolli
ng(window).mean())
smoothed_df.colums = 'AvgTemp'
print(smoothed_df.head(14))
print(smoothed_df.tail())
```

We will obtain the following output:

```
         AvgTemp
Year
1902         NaN
1903         NaN
1904         NaN
1905         NaN
1906         NaN
1907         NaN
1908         NaN
1909         NaN
1910         NaN
1911   17.401761
1912   17.398872
1913   17.394177
1914   17.388443
1915   17.358825
         AvgTemp
Year
2006   19.531170
2007   19.583102
2008   19.581256
2009   19.584580
2010   19.654919
```

Figure 7: 10-year moving average temperatures

Notice that the first 9 samples are **`NaN`**, which is because of the size of the moving average filter window. The window size is 10, hence, 9 (10-1) samples are required to generate the first average, and thus the first 9 samples are **`NaN`**. There are additional options to the **`rolling()`** method that can extend the values to the left or right, or allow the early values to be based on fewer points. In this case, we'll just filter them out:

```
# filter out the NaN values
```

```
smoothed_df =
smoothed_df[smoothed_df['AvgTemp'].notnu
ll()]
# quick plot to understand what we have
so far
fig, ax = plt.subplots()
ax.scatter(smoothed_df.index,
smoothed_df['AvgTemp'])
plt.show()
```

The output will be as follows:

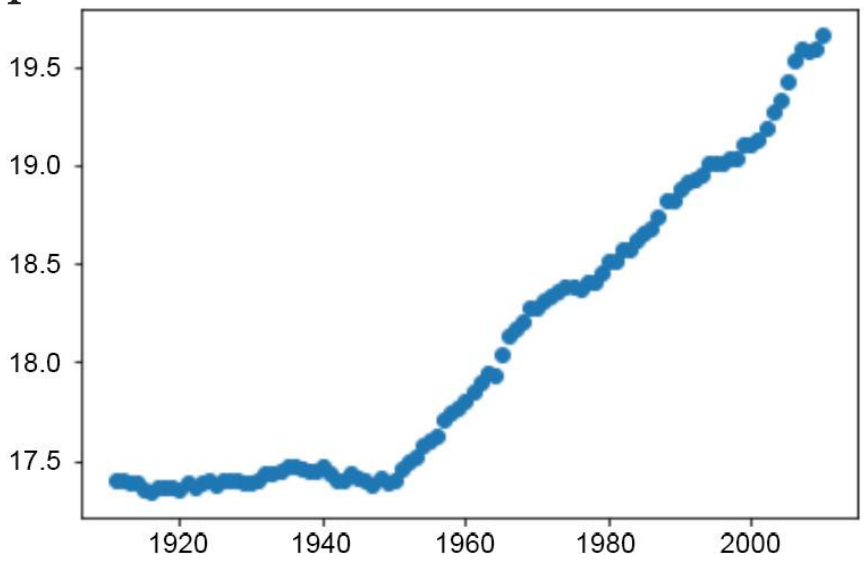

Figure 8: Visualization of preprocessed temperature data

8. Finally, plot the measurements by year along with the moving average signal:

```
fig = plt.figure(figsize=(10, 7))
ax = fig.add_axes([1, 1, 1, 1]);
# Raw data
raw_plot_data = df[df.Year > 1901]
ax.scatter(raw_plot_data.Year, \
           raw_plot_data.RgnAvTemp, \
           label = 'Raw Data', c =
'blue', s = 1.5)
```

```
# Annual averages
annual_plot_data = df_group_year\
                   .filter(items =
smoothed_df.index, axis = 0)
ax.scatter(annual_plot_data.index, \
           annual_plot_data.AvgTemp, \
           label = 'Annual average', c =
'k')
# Moving averages
ax.plot(smoothed_df.index,
smoothed_df.AvgTemp, \
        c = 'r', linestyle = '--', \
        label = f'{window} year moving
average')
ax.set_title('Mean Air Temperature
Measurements', fontsize = 16)
# make the ticks include the first and
last years
tick_years = [1902] + list(range(1910,
2011, 10))
ax.set_xlabel('Year', fontsize = 14)
ax.set_ylabel('Temperature ($^\circ$C)',
fontsize = 14)
ax.set_xticks(tick_years)
ax.tick_params(labelsize = 12)
ax.legend(fontsize = 12)
plt.show()
```

The output will be as follows:

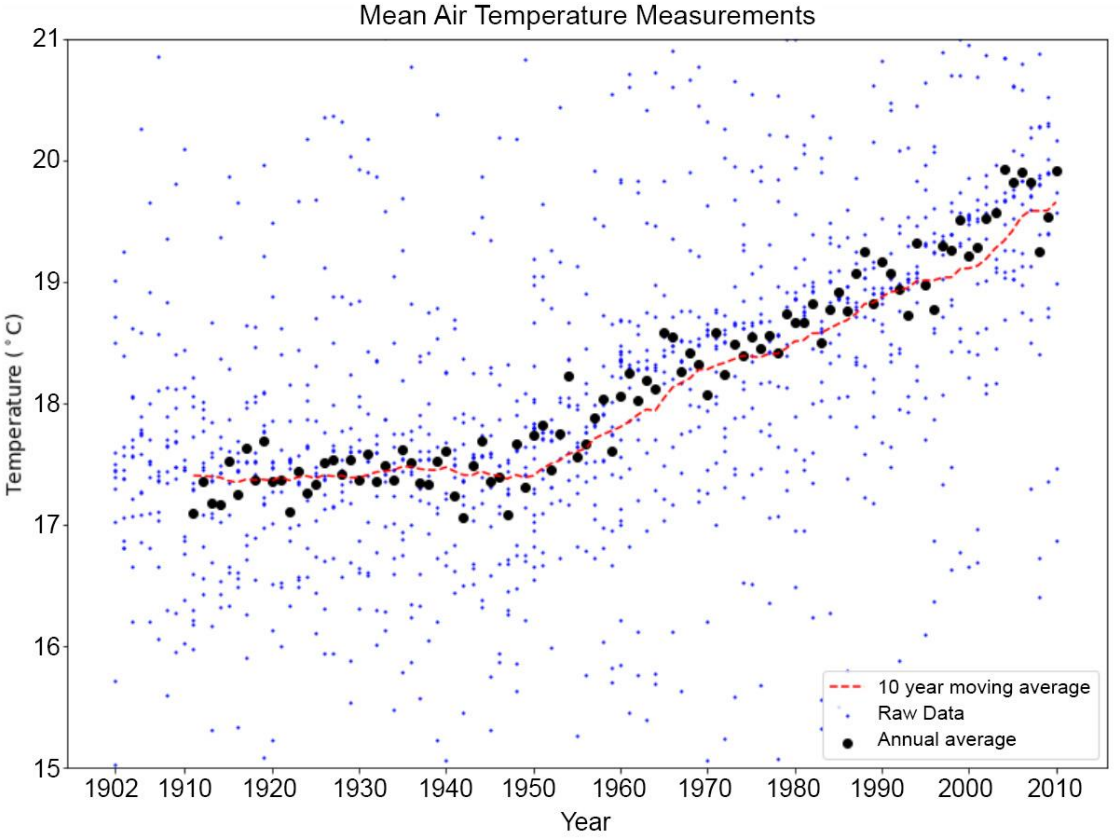

Figure 9: Annual average temperature overlaid on the 10-year moving average

9. We can improve the plot by focusing on the part we are most interested in, the annual average values, by adjusting the *y* scale. This is an important aspect of most visualizations in that the scale should be optimized to convey the most information to the reader:

```
fig = plt.figure(figsize=(10, 7))
ax = fig.add_axes([1, 1, 1, 1]);
# Raw data
raw_plot_data = df[df.Year > 1901]
ax.scatter(raw_plot_data.Year,
raw_plot_data.RgnAvTemp, \
           label = 'Raw Data', c =
'blue', s = 1.5)
```

```
# Annual averages
annual_plot_data = df_group_year\
                   .filter(items =
smoothed_df.index, axis = 0)
ax.scatter(annual_plot_data.index,
annual_plot_data.AvgTemp, \
           label = 'Annual average', c =
'k')
# Moving averages
ax.plot(smoothed_df.index,
smoothed_df.AvgTemp, c = 'r', \
        linestyle = '--', \
        label = f'{window} year moving
average')
ax.set_title('Mean Air Temperature
Measurements', fontsize = 16)
# make the ticks include the first and
last years
tick_years = [1902] + list(range(1910,
2011, 10))
ax.set_xlabel('Year', fontsize = 14)
ax.set_ylabel('Temperature ($^\circ$C)',
fontsize = 14)
ax.set_ylim(17, 20)
ax.set_xticks(tick_years)
ax.tick_params(labelsize = 12)
ax.legend(fontsize = 12)
plt.show()
```

The final plot should appear as follows:

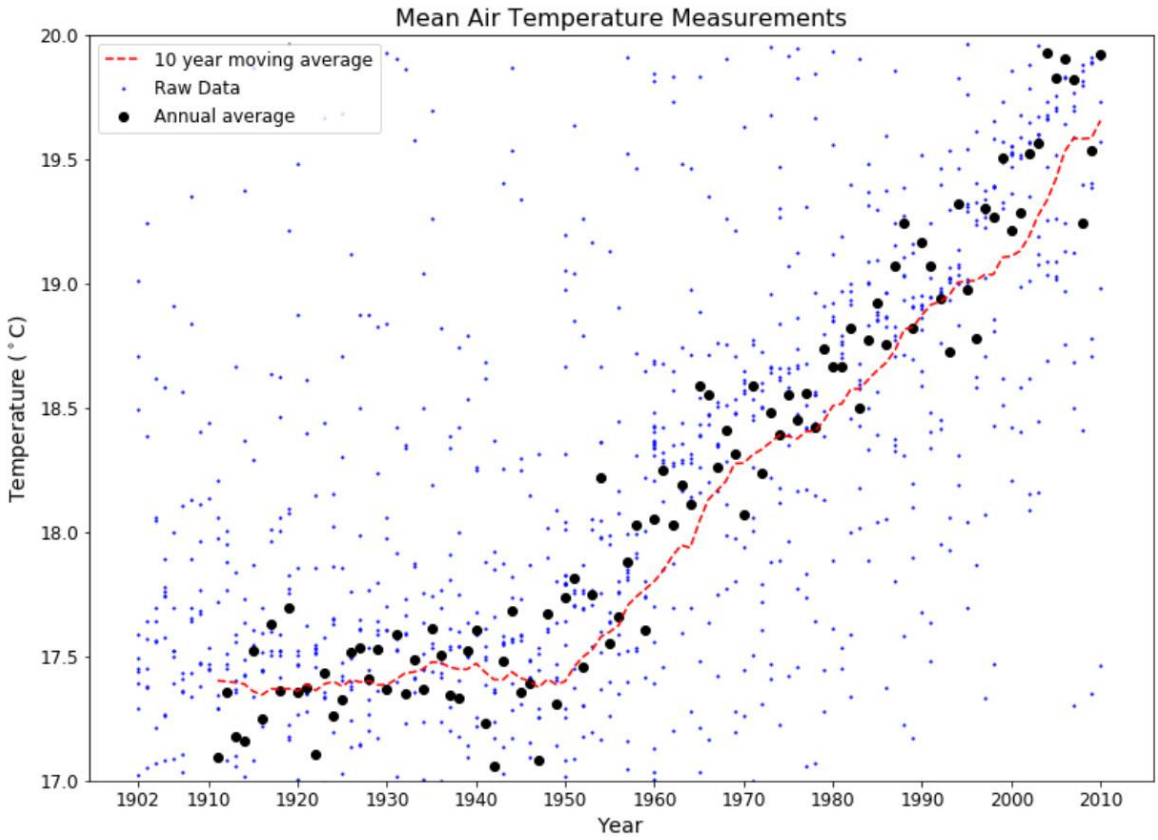

Figure 10: Final plot of raw data, annual averages, and smoothed data

Looking at *Figure 10*, we can immediately make a few interesting observations. First, the temperature remained relatively consistent from the year 1902 to about 1950, after which there is an increasing trend through to the end of the data. Second, there is scatter or noise in the measurements, even after averaging within each year. Third, there appears to be a shift at 1960, which might represent a change in measurement methods or some other factor; we might want to follow up with the business team to understand this more fully.

Finally, note that the moving average values tend to be to the right of the raw data during periods in which there are trends. This is a direct result of the default parameters in the `rolling()` method; each moving average value is the average of 9 points to the left and the current point.

Linear Regression

We will start our investigation into regression models with the selection of a linear model. Linear models, while being a great first choice due to their intuitive nature, are also very powerful in their predictive power, assuming datasets contain some degree of linear or polynomial relationship between the input features and values. The intuitive nature of linear models often arises from the ability to view data as plotted on a graph and observe a trending pattern in the data with, say, the output (the *y*-axis value for the data) trending positively or negatively with the input (the *x*-axis value). The fundamental components of linear regression models are also often learned during high school mathematics classes. You may recall that the equation of a straight line is defined as follows:

$$y = \beta_0 + \beta_1 * x$$

Figure 11: Equation of a straight line

Here, *x* is the input value and *y* is the corresponding output or predicted value. The parameters of the model are the slope of the line (the change in the *y* values divided by the change in *x*, also called the gradient), noted by β_1 in the equation, as well as the *y*-intercept value, β_1, which indicates where the line crosses the *y* axis. With such a model, we can provide values for the β_1 and β_0 parameters to construct a linear model.

For example, $y = 1 + 2 * x$ has a slope of 2, indicating that the changes in the *y* values are at a rate of twice that of *x*; the line crosses the *y* intercept at 1, as you can see in the following diagram:

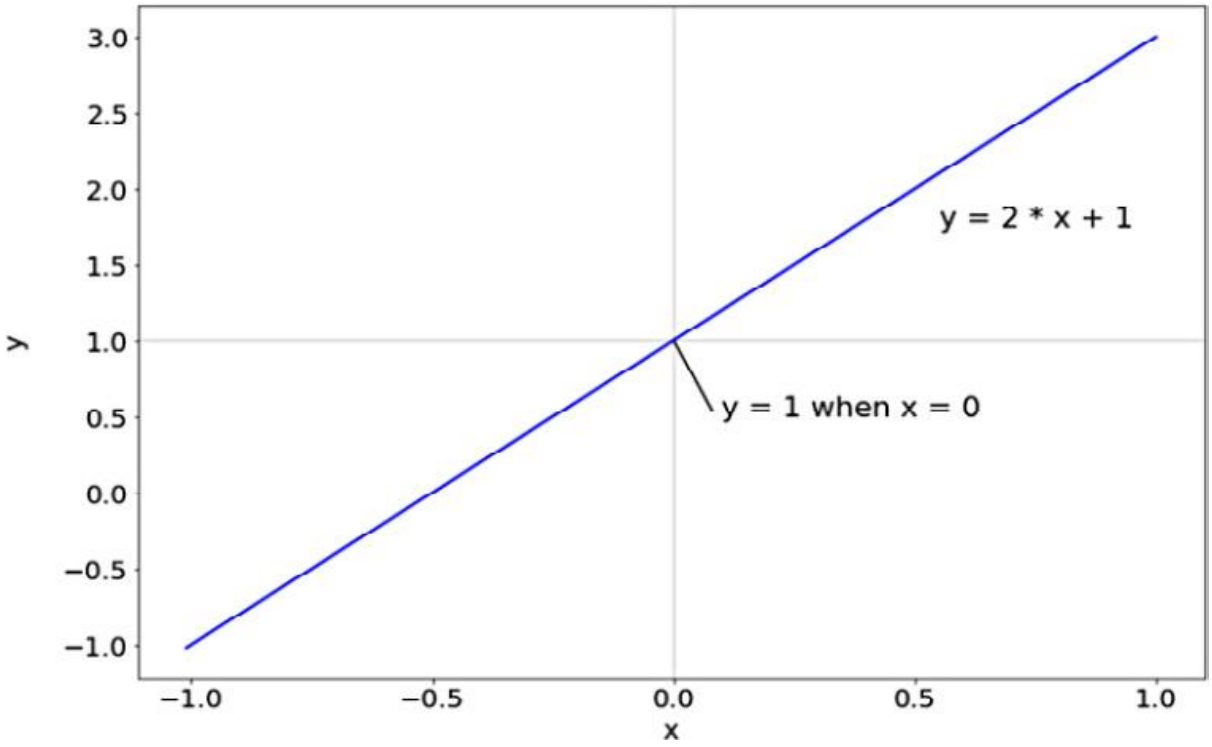

Figure 12: Parameters of a straight line and linear model

So, we have an understanding of the parameters that are required to define a straight line, but this isn't really doing anything particularly interesting. We just dictated the parameters of the model to construct a line. What we want to do is take a dataset and construct a model that best describes a dataset. In terms of the previous section, we want to choose the model architecture as a linear model, and then train the model to find the best values of β_0 and β_1. As mentioned before, this dataset needs to have something that approximates a linear relationship between the input features and output values for a linear model to be a good choice.

Least Squares Method

Many of the various techniques used in machine learning actually significantly pre-date the use of machine learning as a description. Some embody elements of statistics, and others have been used in the sciences to "fit" data for a very long time. The least squares method of finding the equation of a straight line that best represents a set of data is one of these, originally created in the early 1800s. The method can be used to illustrate many key ideas of the supervised learning of regression models, and so we'll start with it here.

The least squares method focuses on minimizing the square of the error between the predicted *y* values and the actual *y* values. The idea of minimizing an error is fundamental in machine learning and is the basis for essentially all learning algorithms.

Although simple linear regression using the least squares method can be written down as simple algebraic expressions, most packages (like scikit-learn) will have more general optimization methods "under the hood."

The Scikit-Learn Model API

The scikit-learn API uses a similar code pattern irrespective of the type of model being constructed. The general flow is:

1. Import the class for the model type you want to use.
   Here, we will use `from sklearn.linear_model import LinearRegression`.
2. Instantiate an instance of the model class. This is where hyperparameters are set. For simple linear regression, we can use defaults.
3. Use the `fit` method with the *x* and *y* data we want to model.
4. Inspect the results, get metrics, and then visualize them.

Let's use this workflow to create a linear regression model in the next exercise.

Exercise 2: Fitting a Linear Model Using the Least Squares Method

In this exercise, we will construct our first linear regression model using the least squares method to visualize the air temperatures over a yearly timeframe and evaluate the performance of the model using evaluation metrics:

Note

We will be using the same **`synth_temp.csv`** *dataset as in Exercise 1: Plotting Data with a Moving Average.*

1. Import the **`LinearRegression`** class from the **`linear_model`** module of scikit-learn, along with the other packages we need:
```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import
LinearRegression
```
2. Load the data. For this exercise, we'll use the same synthetic temperature data as was used previously:
```
# load the data
df =
pd.read_csv('../Datasets/synth_temp.csv'
)
```
3. Repeat the preprocessing of the data from before:
```
# slice 1902 and forward
df = df.loc[df.Year > 1901]
# roll up by year
df_group_year =
df.groupby(['Year']).agg({'RgnAvTemp' :
'mean'})
df_group_year.head(12)
# add the Year column so we can use that
in a model
```

```
df_group_year['Year'] =
df_group_year.index
df_group_year = \
df_group_year.rename(columns =
{'RgnAvTemp' : 'AvTemp'})
df_group_year.head()
```

The data should appear as follows:

Year	AvTemp	Year
1902	17.385044	1902
1903	17.222163	1903
1904	17.217215	1904
1905	17.817502	1905
1906	17.386445	1906

Figure 13: Data after preprocessing

4. Instantiate the **`LinearRegression`** class. Then, we can fit the model using our data. Initially, we will just fit the temperature data to the years. In the following code, note that the method requires the *x* data to be a 2D array and that we are passing only the year. We also need to use the **`reshape`** method and, in the `(-1, 1)` parameters, `-1` means that "the value is inferred from the length of the array and remaining dimensions":

```
# construct the model and inspect
results
linear_model =
LinearRegression(fit_intercept = True)
```

```
linear_model.fit(df_group_year['Year'].v
alues.reshape((-1, 1)), \
                 df_group_year.AvTemp)
print('model slope = ',
linear_model.coef_[0])
print('model intercept = ',
linear_model.intercept_)
r2 =
linear_model.score(df_group_year['Year']
\
                           .values.reshape(
(-1, 1)), \
                           df_group_year.Av
Temp)
print('r squared = ', r2)
```

Note

Refer to the following link for more reading on scikit-learn: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

The output will be as follows:

```
model slope =  0.02352237024970654
model intercept =  -27.88736502793287
r squared =  0.8438499294671093
```

Figure 14: Results from using the fit method

Note the use of the **`score`** method, which is a method of the **`model`** object, to obtain the r^2 value. This metric, called the coefficient of determination, is a widely used metric for linear regression. The closer r^2 is to 1, the more closely our model is predicting the data.

There are multiple formulas that can be used to compute r^2. Here is an example:

$$r^2 = 1 - \frac{\Sigma(y_i - \hat{y})^2}{\Sigma(y_i - \bar{y})^2}$$

Figure 15: Calculation of r2

From *Figure 15*, you can get some understanding of r^2 by noting that the numerator sums the errors from the predictions, while the denominator sums the variation of the data from the mean. Thus, r^2 increases as the prediction errors get smaller. It's important to emphasize here that r^2 is just a measure of "goodness of fit"—in this case, how well a simple straight line fits the given data. In more complex, real-world supervised learning problems, we would use a more robust approach to optimize the model and choose the best/final model. In particular, in general, we evaluate the model on data *not* used to train it, because evaluating it on the training data would give an overly optimistic measure of performance.

5. To visualize the results, we need to pass some data to the **`predict`** method of the model. A simple way to do that is to just reuse the data we used to fit the model:

```
# generate predictions for visualization
pred_X = df_group_year.loc[:, 'Year']
pred_Y =
linear_model.predict(df_group_year['Year
']\
                                .values.res
hape((-1, 1)))
```

6. Now, we have everything we need to visualize the result:

```
fig = plt.figure(figsize=(10, 7))
ax = fig.add_axes([1, 1, 1, 1]);
# Raw data
raw_plot_data = df[df.Year > 1901]
ax.scatter(raw_plot_data.Year,
raw_plot_data.RgnAvTemp, \
           label = 'Raw Data', c =
'red', s = 1.5)
# Annual averages
ax.scatter(df_group_year.Year,
df_group_year.AvTemp, \
           label = 'Annual average', c =
'k', s = 10)
# linear fit
ax.plot(pred_X, pred_Y, c = "blue",
linestyle = '-.', \
        linewidth = 4, label = 'linear
fit')
ax.set_title('Mean Air Temperature
Measurements', fontsize = 16)
# make the ticks include the first and
last years
```

```
tick_years = [1902] + list(range(1910,
2011, 10))
ax.set_xlabel('Year', fontsize = 14)
ax.set_ylabel('Temperature ($^\circ$C)',
fontsize = 14)
ax.set_ylim(15, 21)
ax.set_xticks(tick_years)
ax.tick_params(labelsize = 12)
ax.legend(fontsize = 12)
plt.show()
```

The output will be as follows:

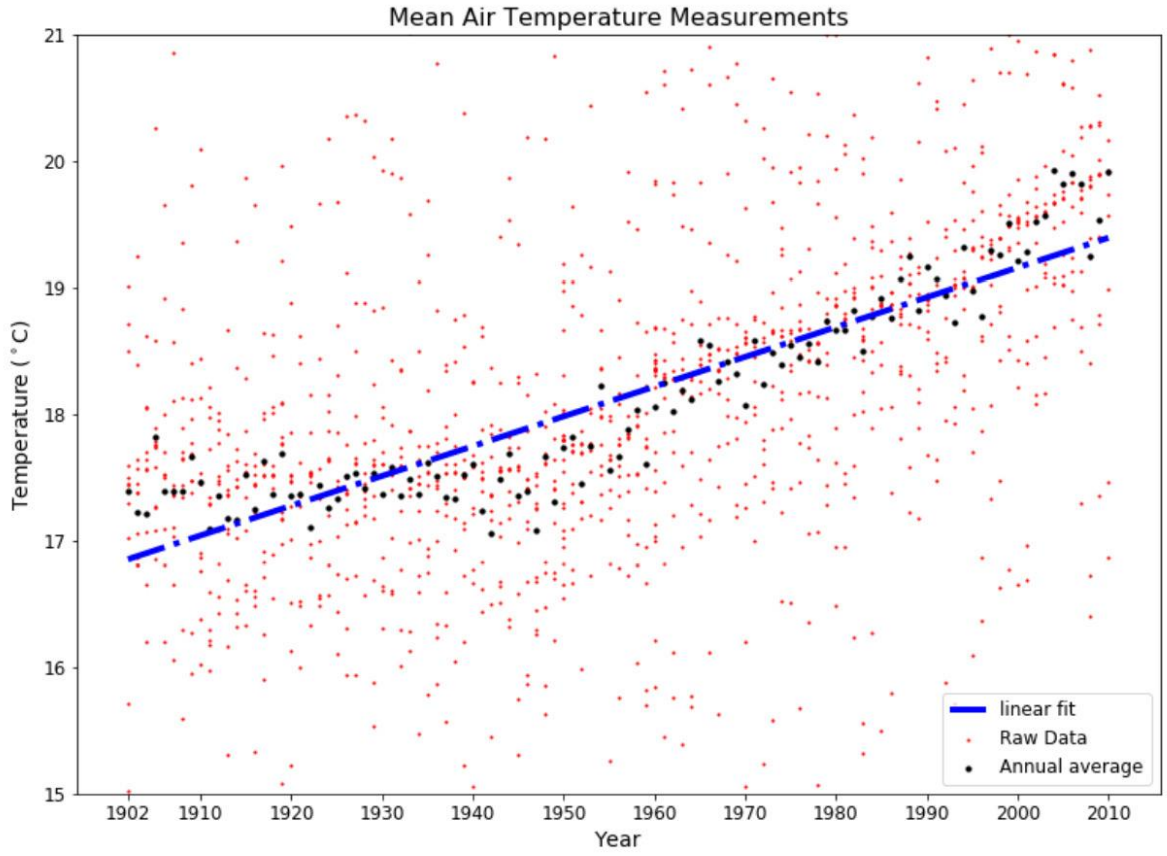

Figure 6: Linear regression – a first simple linear model

From *Figure 6*, it's evident that a straight line isn't a very good model of the data. We'll return to this issue after an activity.

We have seen how to load in some data, import the **`LinearRegression`** class from scikit-learn, and use the **`fit`**, **`score`**, and **`predict`** methods to construct a model, look at a performance metric, and then visualize the results. Along the way, we introduced the least squares method, gave some of the mathematical background, and showed how some of the calculations work.

We saw that for our synthetic temperature data, a linear model doesn't fit the data all that well. That's okay. In most cases, it is good practice to generate a baseline model early on in the project to serve as a benchmark against which the performance of more sophisticated models can be compared. So we can consider the linear model we developed here to be a naïve baseline model.