Case Study 2: Predicting Future and Past Temperatures Using Simple Linear Regression

If we were given a collection of numeric values representing an independent variable and a dependent variable, simple linear regression describes the relationship between these variables with a straight line, known as the regression line. The points along the regression line (in two dimensions) like those shown in the preceding graph can be calculated with the equation:

*y* = *mx* + *b*

where

* *m* is the line’s slope,
* *b* is the line’s intercept with the *y*-axis (at *x* = 0),
* *x* is the independent variable (the date in this example), and
* *y* is the dependent variable (the temperature in this example).

In simple linear regression, *y* is the *predicted value* for a given *x*.

In this example, we’ll

* use a *scikit-learn estimator* to reimplement the simple linear regression
* use Seaborn’s scatterplot function to plot the data and Matplotlib’s plot function to display the regression line, then
* use the scikit-learn estimator to make predictions.

Data Description

The data represent the 1895 through 2018 January average high temperatures in New York City. We will use this data to predict future average January high temperatures and to estimate the average January high temperatures for years preceding 1895. The observations are *ordered* by year and *one* observation per time, such as the average of the January high temperatures in New York City for a particular year.

For your convenience, we provide the temperature data in a CSV file named Lab 1 dataset.CSV. So, get your environment ready and save the CSV file in a folder called data in your user account.

Loading the Average High Temperatures into a DataFrame

Let’s see how the data looks like, here are a few samples:

| Date | Value | Anomaly |
| --- | --- | --- |
| 189501 | 34.2 | -3.2 |
| 189601 | 34.7 | -2.7 |
| 189701 | 35.5 | -1.9 |
| 189801 | 39.6 | 2.2 |
| 189901 | 36.4 | -1 |
| 190001 | 37.4 | 0 |
| 190101 | 37 | -0.4 |
| 190201 | 35 | -2.4 |
| 190301 | 35.5 | -1.9 |
| 190401 | 29.8 | -7.6 |
| 190501 | 33.7 | -3.7 |
| 190601 | 42.3 | 4.9 |
| 190701 | 40.5 | 3.1 |
| 190801 | 38.3 | 0.9 |
| 190901 | 39.6 | 2.2 |
| 191001 | 36.1 | -1.3 |
| 191101 | 40.7 | 3.3 |
| 191201 | 29.3 | -8.1 |
| 191301 | 46.9 | 9.5 |
|  |  |  |

This data contains three columns per observation:

* Date—A value of the form 'YYYYMM’ (such as '201801'). MM is always 01 because we downloaded data for only January of each year.
* Value—A floating-point Fahrenheit temperature.
* Anomaly—The difference between the value for the given date and the average values for all dates. We do not use the Anomaly value in this example, so we’ll ignore it.

Let’s load the data from Lab 1 dataset.csv, rename the 'Value' column to 'Temperature', remove 01 from the end of each date value using floordiv function and display a few data samples after the changes:

In [1]: import pandas as pd

In [2]: nyc = pd.read\_csv('Lab 1 dataset.csv')

In [3]: nyc.columns = ['Date', 'Temperature', 'Anomaly']

In [4]: nyc.Date = nyc.Date.floordiv(100)

In [5]: nyc.head(3)

Out[5]:

Date Temperature Anomaly

1895 34.2 -3.2

1896 34.7 -2.7

1897 35.5 -1.9

Splitting the Data for Training and Testing

Let’s split the data into training and testing sets. Once again, we used the keyword argument random\_state for reproducibility:

In [6]: from sklearn.model\_selection import train\_test\_split

In [7]: X\_train, X\_test, y\_train, y\_test = train\_test\_split(

...: nyc.Date.values.reshape(-1, 1), nyc.Temperature.values,

...: random\_state=11)

...:

Reshape the data into a two-dimensional array.

The expression nyc.Date returns the Date column’s Series, and the Series’ values attribute returns the NumPy array containing that Series’ values. To transform this one-dimensional array into two dimensions, we call the array’s reshape method. Normally, two arguments are the precise number of rows and columns. However, the first argument -1 tells reshape to *infer* the number of rows, based on the number of columns (1) and the number of elements (124) in the array. The transformed array will have only one column, so reshape infers the number of rows to be 124, because the only way to fit 124 elements into an array with one column is by distributing them over 124 rows.

We can confirm the 75%–25% train-test split by checking the shapes of X\_train and X\_test:

In [8]: X\_train.shape

Out[8]: (93, 1)

In [9]: X\_test.shape

Out[9]: (31, 1)

Training the Model

Let’s train a LinearRegression estimator:

In [10]: from sklearn.linear\_model import LinearRegression

In [11]: linear\_regression = LinearRegression()

In [12]: linear\_regression.fit(X=X\_train, y=y\_train)

Out[12]: LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None,

normalize=False)

After training the estimator, fit returns the estimator.

To find the best fitting regression line for the data, the LinearRegression estimator iteratively adjusts the slope and intercept values to minimize the sum of the squares of the data points’ distances from the line.

Now, we can get the slope and intercept used in the *y = mx + b* calculation to make predictions. The slope is stored in the estimator’s coeff\_ attribute (*m* in the equation) and the intercept is stored in the estimator’s intercept\_ attribute (*b* in the equation):

In [13]: linear\_regression.coef\_

Out[13]: array([0.01939167])

In [14]: linear\_regression.intercept\_

Out[14]: -0.30779820252656265

We’ll use these later to plot the regression line and make predictions for specific dates.

Testing the Model

Let’s test the model using the data in X\_test and check some of the predictions throughout the dataset by displaying the predicted and expected values for every fifth element:

In [15]: predicted = linear\_regression.predict(X\_test)

In [16]: expected = y\_test

In [17]: for p, e in zip(predicted[::5], expected[::5]):

...: print(f'predicted: {p:.2f}, expected: {e:.2f}')

...:

predicted: 37.86, expected: 31.70

predicted: 38.69, expected: 34.80

predicted: 37.00, expected: 39.40

predicted: 37.25, expected: 45.70

predicted: 38.05, expected: 32.30

predicted: 37.64, expected: 33.80

predicted: 36.94, expected: 39.70

Predicting Future Temperatures and Estimating Past Temperatures

Let’s use the coefficient and intercept values to predict the January 2019 average high temperature and to estimate what the average high temperature was in January of 1890. The lambda in the following snippet implements the equation for a line

*y* = *mx* + *b*

using the coef\_ as *m* and the intercept\_ as *b*.

In [18]: predict = (lambda x: linear\_regression.coef\_ \* x +

...: linear\_regression.intercept\_)

...:

In [19]: predict(2019)

Out[19]: array([38.84399018])

In [20]: predict(1890)

Out[20]: array([36.34246432])

Visualising the Dataset with the Regression Line

Next, let’s create a scatter plot of the dataset using Seaborn’s scatterplot function and Matplotlib’s plot function. First, use scatterplot with the nyc DataFrame to display the data points:

In [21]: import seaborn as sns

In [22]: axes = sns.scatterplot(data=nyc, x='Date', y='Temperature',

...: hue='Temperature', palette='winter', legend=False)

...:

The keyword arguments are:

* data, which specifies the DataFrame (nyc) containing the data to display.
* x and y, which specify the names of nyc’s columns that are the source of the data along the *x*- and *y*-axes, respectively. In this case, x is the 'Date' and y is the 'Temperature'. The corresponding values from each column form *x-y* coordinate pairs used to plot the dots.
* hue, which specifies which column’s data should be used to determine the dot colors. In this case, we use the 'Temperature' column. Color is not particularly important in this example, but we wanted to add some visual interest to the graph.
* palette, which specifies a Matplotlib color map from which to choose the dots’ colors.
* legend=False, which specifies that scatterplot should not show a legend for the graph—the default is True, but we do not need a legend for this example.

Let’s scale the y-axis range of values so you’ll be able to see the linear relationship better once we display the regression line:

In [23]: axes.set\_ylim(10, 70)

Out[23]: (10, 70)

Next, let’s display the regression line. First, create an array containing the minimum and maximum date values in nyc.Date. These are the *x*-coordinates of the regression line’s start and end points:

In [24]: import numpy as np

In [25]: x = np.array([min(nyc.Date.values), max(nyc.Date.values)])

Passing the array x to the predict lambda from snippet [16] produces an array containing the corresponding predicted values, which we’ll use as the *y*-coordinates:

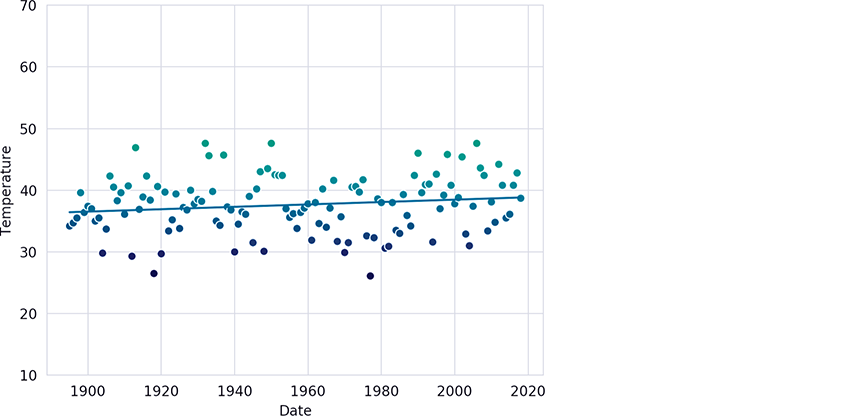
In [26]: y = predict(x)

Finally, we can use Matplotlib’s plot function to plot a line based on the x and y arrays, which represent the *x*- and *y-*coordinates of the points, respectively:

In [27]: import matplotlib.pyplot as plt

In [28]: line = plt.plot(x, y)

The resulting scatterplot and regression line are shown below.



Overfitting/Underfitting

When creating a model, a key goal is to ensure that it is capable of making accurate predictions for data it has not yet seen. Two common problems that prevent accurate predictions are overfitting and underfitting:

* Underfitting occurs when a model is too simple to make predictions, based on its training data. For example, you may use a linear model, such as simple linear regression, when in fact, the problem really requires a non-linear model. For example, temperatures vary significantly throughout the four seasons. If you’re trying to create a general model that can predict temperatures year-round, a simple linear regression model will underfit the data.
* Overfitting occurs when your model is too complex. The most extreme case, would be a model that memorizes its training data. That may be acceptable if your new data looks *exactly* like your training data, but ordinarily that’s not the case. When you make predictions with an overfit model, new data that matches the training data will produce perfect predictions, but the model will not know what to do with data it has never seen.