Hyperiondev

TASK

Supervised Learning – Multiple Linear Regression

Visit our website

Introduction

WELCOME TO THE SUPERVISED LEARNING – MULTIPLE LINEAR REGRESSION TASK!

In the previous tasks, we explored the relationship between one explanatory variable and the response or dependent variable. In many cases, you will be interested in more than a single input variable. Thus, we need to learn about multiple linear regression, a special case of simple linear regression that adds more terms to the regression model.

MULTIPLE LINEAR REGRESSION

In the last task, we saw how simple linear regression can shed light on the relationship between two variables. We looked only at the impact of a single-independent variable on a single-input variable. The example case, however, had two independent variables: TV and social media. Multiple linear regression allows us to model the impact of both variables on the outcome variable at the same time.

Modelling multiple variables is quite useful. A simple linear regression approach may overestimate or underestimate the impact of a variable on the outcome, because it does not have any information about the impact of other variables. Imagine, for instance, that our social media and billboards budget were increased simultaneously, and sales increased shortly thereafter. Our simple regression models would attribute the entire increase in sales to the single variable they were modelling, which would be incorrect. A multiple linear regression model could make a less biased prediction of how much each type of advertisement contributes to sales.

The extension of simple linear regression is fairly straightforward. Instead of a function for **Y** with only one coefficient, the function has coefficients for each variable. In the case of two variables, the formula takes the form:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2$$

Note that the coefficients ($\boldsymbol{\beta_i}$) will not just be the same values as independent simple linear regression models would return. This extended model will adjust each value according to the relative contribution of each variable.

Take note:

Over time, several myths have emerged surrounding machine learning, such as "machine learning is just summarising data" or "learning algorithms just discover correlations between events". Probably one of the most prominent is that "machine learning can't predict unseen events". This line of thought goes thus: if something has never happened before, its predicted probability must be zero. Right?

On the contrary, machine learning is the art of predicting rare events with high accuracy. If A is one of the causes of B, and B is one of the causes of C, A can lead to C, even if we've never seen it happen before. A practical example of the predictive powers of machine learning is the spam filter in your email inbox. Every day, spam filters correctly flag freshly concocted spam emails, based on emails you marked as spam earlier and other predictor data.

APPLICATION

Both simple and multiple linear regression can be performed very simply using **sklearn**. Provided that your data set is properly pre-processed, **sklearn** infers on its own whether your input has one or multiple features. Fitting a multiple regression model looks like this:

```
# Fit a multiple regression model
multiple_model = LinearRegression()
multiple_model.fit(X, y)
```

Let's apply this to some data on the impact of TV, radio, and newspaper advertising on sales. This data set, **Advertising.csv**, is also in your lesson folder if you want to try it for yourself.

Multiple linear regression applied to this data set returns the coefficients `0.045`, `0.189`, and -`0.001`.

To see what these coefficients say about the variables, it helps to see them in the context of the formula for **Y**:

sales = 2.94 + 0.045(TV) + 0.189(radio) – 0.001(newspaper)

One thing this shows is that TV and radio advertising has a positive impact on sales, but newspaper advertising has an impact that is close to zero, and even a bit below it. Negative coefficients mean that the variables have an opposite relationship: as one increases, the other one decreases. When comparing the

coefficients for TV and radio, we see that the coefficient for radio is larger than that for TV. This means that radio contributes more to total sales than TV does.

Based on these findings, we may recommend a focus on radio advertising. We might also recommend that the newspaper advertising budget be scrapped or diminished.

TRAINING AND TEST DATA IN MACHINE LEARNING

When you're teaching a student arithmetic summation, you want to start by teaching them the pattern of summation, and then test whether they can apply that pattern. You will show them that 1+1=2, 4+5=9, 2+6=8, and so forth. If the student memorises all the examples, they could answer the question 'what is 2+6?' without understanding summation. To test whether they have recovered not just the facts but the pattern behind the facts, you need to test them on numbers that you have not exposed them to directly. For example, you might ask them 'what is 4+9?'

This intuition forms the motivation behind training and testing data in machine learning. We create a model (e.g. regression) based on some data that we have, but we do not use all of our data: we hide some data samples from the model and then test our model on the hidden data samples to confirm that the model is making valid predictions as opposed to reiterating what was given to it in the training data set.

A **training set** is the actual data set that we use to train the model. The model sees and learns from this data. A **test set** is a set of withheld examples used only to assess the performance of a model. Although the best ratio depends on how much data is available, a common split is a ratio of 80:20. For example, 8,000 training examples and 2,000 test cases. **Sklearn** allows us to do this quite easily:

```
# Import the train_test_split functionality from sklearn
from sklearn.model_selection import train_test_split

# Pass your x and y variables in the function and get the train and test
# sets for the X and y variables (4 variables).
# The test_size parameter determines how the data is split, 0.25 means 25%
# of the data will be in the test set.

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
```

One thing to watch out for when dividing the data is that the test and training set should not be systematically different. If the total data set is ordered in some way, for instance by alphabet or by the time of collection, the test set might contain different kinds of instances from the training set. If that occurs, the model's performance will be poor, not because it didn't learn from the data effectively, but because it is being tested on a task that differs from the one it was trained for. For this purpose, it is customary to investigate the distribution of labels in your data and make sure they are similar across your training and test data. In the example below, notice that there are no `0` samples in the test set.

```
# Data sample
X = [1,2,3,4,2,6,7,8,6,7]
y = [0,0,0,0,0,1,1,1,1,1]

# Split data
X_train, X_test, y_train,y_test = train_test_split(X, y, test_size = 0.2,
shuffle=False)

# Compare train and test sets for y
print("y_train {}".format(y_train))
print("y_test {}".format(y_test))
```

```
Output:
>> y_train [0, 0, 0, 0, 0, 1, 1, 1]
>> y_test [1, 1]
```

Notice there are no `0` labels in the test set (`y_test`). This can largely be solved by using `shuffle=True` as a parameter in `train_test_split`. This means that the labels are distributed randomly between the training and test sets, so it is less likely that the training and testing data would be systematically different.

```
Output:
>>y_train [1, 0, 0, 1, 0, 1, 0, 1]
>>y_test [0, 1]
```

Notice there is now a `0` label in the test set. However, there is still the possibility that most samples of one type end up in the training set, without a representative proportion in the test set. This could also result in lower performance than expected. Consider the case where we set `test_set=0.4`, and we get the following:

```
Output:
```

```
>>y_train [0, 1, 1, 1, 0, 1]
>>y_test [0, 0, 0, 1]
```

Notice the labels are not distributed proportionally between `y_test` and `y_train`. Another way to ensure that data is represented equally in both the training and testing data is to make use of the stratify parameter. This ensures that labels are represented in as close to the same proportions as possible in both the training and testing data.

```
# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.5,
shuffle=True, stratify=y)
```

```
Output:
>>y_train [0, 0, 1, 1, 1, 0]
>>y_test [1, 0, 0, 1]
```

Note that the labels are now distributed in the same proportion across the training and test sets.

Take note:

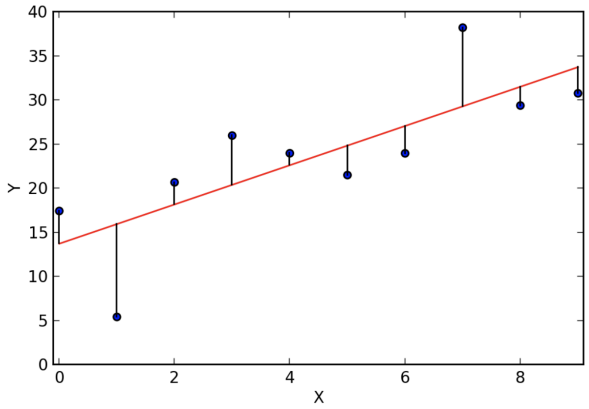
You need to be using `sklearn` version 0.17 or higher to make use of the stratify parameter.

TRAINING AND TEST ERROR

We've discussed before how a regression model is only an approximation of the data. The model predicts values that are close to the observed training values, in hopes of making good predictions on unseen data. The difference between the actual values and the predictions is called the *error*.

The training data set error and the test data set error can be obtained by comparing the predictions to the known, true labels (the gold standard). The test error value is more important as it is the more reliable assessment of the value of the model. After all, we want to use our model on unknown data points, as opposed to applying it to cases for which we already have the actual outcome. In most cases, the training error value will also be lower than the test error value.

There are many different ways to measure the error. As mentioned, the error is the difference between observed and predicted values, as in this plot:



Here, we have observed data (**Y**) in dark blue and prediction of a regression model (**y_pred**) along the red line. The error is depicted by vertical black lines. There are a number of different ways one can aggregate the error to get a final score for the model. Two common ones are the root mean squared error (RMSE) and R-squared (R^2).

Extra resource

Read more about the **R-squared metric** to learn how to interpret this metric and the limitations you should be aware of when using it to evaluate regression models.

Instructions

- First, open the accompanying **multiple_linear_regression.ipynb** notebook to see a code example of multiple regression in Python.

Practical Task

Follow these steps:

- Create a Jupyter Notebook called **diabetes_regression.ipynb**.
- Read **diabetes_dirty.csv** into your Jupyter Notebook.
- The **diabetes_dirty.csv** aims to predict a person's progression in the condition with respect to various attributes about them.
- Differentiate between the independent variables and the dependent variable, and assign them to variables X and Y.
- Generate training and test sets comprising 80% and 20% of the data respectively.
- Investigate the necessity for scaling or normalization of the data. Employ `MinMaxScaler` and `StandardScaler` if necessary. Fit these scalers on the training set and apply the fitted scalers to transform both the training and test sets accordingly.
- Generate a multiple linear regression model using the **training set**. Use all of the independent variables.
- Print out the intercept and coefficients of the trained model.
- Generate predictions for the **test set**.
- Compute R-squared for your model on the **test set**. You can use `r2_score` from `sklearn.metrics` to obtain this score.
- Ensure your Notebook includes comments about what your code is accomplishing and notes about model outputs such as R-squared.

Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.