</>



David Ziganto Senior Data Scientist ♥ Chicago in LinkedIn GitHub

Linear Regression 101 (Part 2 - Metrics) ② 5 minute read

 $R^2 = 0.06$ REXTHOR, THE DOG-BEARER I DON'T TRUST LINEAR REGRESSIONS WHEN IT'S HARDER TO GUESS THE DIRECTION OF THE CORRELATION FROM THE SCATTER PLOT THAN TO FIND NEW CONSTELLATIONS ON IT. Introduction

parameters for both univariate and multivariate linear regression.

Now we'll turn our focus to metrics pertaining to our model. We'll look at the following key metrics: 1. Sum of Squared Errors (SSE)

regression. Specifically, we learned key terminology and how to find

We left off last time (Part 1) discussing the basics of linear

4. Adjusted R^2

- To keep things simple, we'll use the univariate baby weight data
- from the previous post and leverage sklearn to find the model
- parameters.

import numpy as np # reproducibility np.random.seed(10) # generate data

babies = range(10) months = np.arange(13)

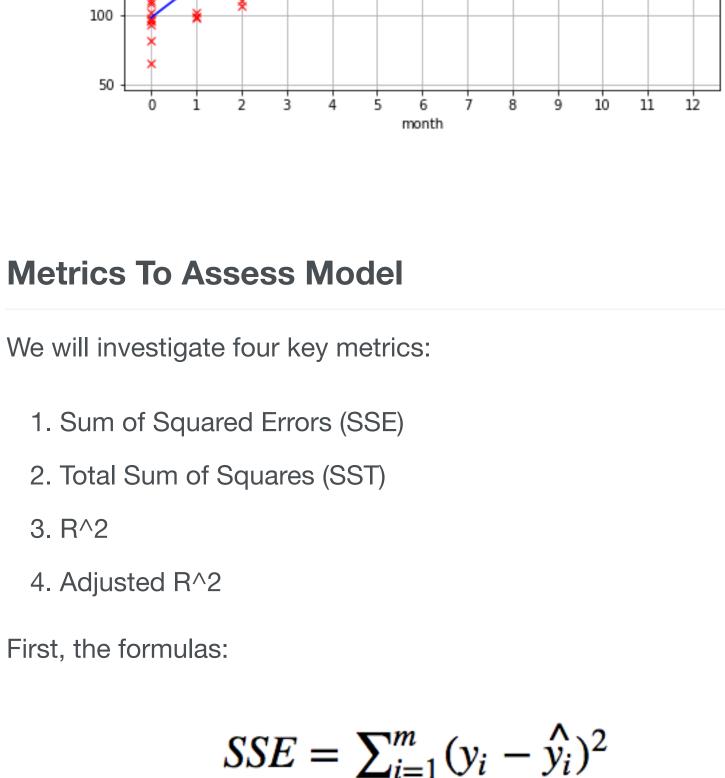
```
for month in months
           for baby in babies]
   month_data = [element[0] for element in data]
   weight_data = [element[1] for element in data]
Let's fit the model. Instead of using the linear algebra code from the
first post, we can use the scikit-learn (sklearn) module to do the
heavy lifting for us. Here's how you do that:
                                                                     </>
   from sklearn.linear_model import LinearRegression
   X = np.array(month_data).reshape(-1,1)
   y = weight_data
   lr = LinearRegression(fit_intercept=True)
   lr.fit(X, y)
```

Technical note: X contains .reshape(-1,1) which creates a

```
you should remember it.
As a reminder, the plot looks like this:
                                    Baby Weights
                baby weight
                linear regression
```

250 250

150



weigths, then m equals 25. Lastly,
$$df_t$$
 is the degrees of freedom of the estimate of the population variance of the dependent variable and df_e is the degrees of freedom of the estimate of the underlying population error variance. If df_t and df_e don't make much sense, don't worry about it. They're simply used to calculate adjusted R^2.

should still get the main idea, though some nuances may be lost.

Without further ado, let's create a class to capture the four key

</>

 $\operatorname{adj} R^2 = 1 - \frac{SSE * df_t}{SST * df_e}$

Keep in mind that *y_i* is the observed target value, *y-hat_i* is the

predicted value, and *y-bar* is the mean value. Here, *m* represents

the total number of observations. For example, if there are 25 baby

 $self._dft = X.shape[0] - 1$ ## degrees of freedom population error variance $self._dfe = X.shape[0] - X.shape[1] - 1$ def sse(self): '''returns sum of squared errors (model vs actual)''' squared_errors = (self.target - self.model.predict(self.date return np.sum(squared_errors)

degrees of freedom population dep. variable variance



Answer: squaring the values makes them all positive. If we didn't

values. No matter how you slice it, you end up with error that looks

square them, then we'd have some positive and some negative

But why not use absolute error instead of squared error?

If you think about what squaring does to large numbers, you'll

realize that we're really penalizing large errors. It's like saying, it's

okay to miss on the close points but don't allow large deviations

SST is a measure of the variance in the target variable. It is

each observation and the target mean.

measured simply as the sum of the squared difference between

R^2 measures how much variance is captured by the model. The

negative values for R^2 but that would require a fitting procedure

other than OLS or non-linear data. Always know your assumptions!

range for Ordinary Least Squares is [0,1]. It is possible to get

between the model and the most distant points. This makes sense

for some datasets but not others. Consider the converse, though. There's a data point or points that deviate from the general trend, causing large squared errors. These

Why is that?

smaller than it is in reality.

(Adjusted R^2) Adjusted R^2 is the same as standard R^2 except that it penalizes models when additional features are added. This naturally leads to the next section about why R^2 is a poor metric to use. Why R² is a Poor Metric

Compare R^2 and Adjusted R^2 min_features = 1

results = sm.OLS(y, X).fit() r_squared.append(results.rsquared) adj_r_squared.append(results.rsquared_adj) **Plot Varying Number of Features** R^2/R^2 adjusted ~ # features

y = generate_random_data(num_rows=200, num_features=1, random_state:

out because of the penalty involved. The big takeaway here is that you cannot compare two linear regression models with differing numbers of features using R^2 alone. It just cannot be done. Adjusted R^2 works, though. Wrap Up

As we'll see in a future post, there are better ways to assess

machine learning models, namely in-sample vs out-of-sample

metrics and cross-validation. More on those in a future post about

Next time we'll dive into linear regression assumptions, pitfalls, and ways to address problems.

Linear Regression

Updated: January 08, 2018

2. Total Sum of Squares (SST) 3. R^2

Here's how I generated the data again:

data = [(month, np.dot(month, 24.7) + 96 + np.random.normal(loc=0,

fake 2D array for fitting. This trick pops up quite frequently so 350

$SST = \sum_{i=1}^{m} (y_i - \bar{y})^2$

 $R^2 = 1 - \frac{SSE}{SST}$

def __init__(self, X, y, model):

self.data = X

def r_squared(self):

def adj_r_squared(self):

self.target = y

self.model = model

statistics about our data.

class Stats:

def sst(self): '''returns total sum of squared errors (actual vs avg(actua avg_y = np.mean(self.target) squared_errors = (self.target - avg_y) ** 2 return np.sum(squared_errors)

'''returns calculated value of adjusted r^2'''

'''returns calculated value of r^2'''

return 1 - self.sse()/self.sst()

anomalous data points are often called outliers, and they can wreak havoc on the performance of your model. Much more on this in the next post.

(SST)

(R²)

example below.

Generate Dummy Data

if random_state:

else:

return data

max_features = 100

adj_r_squared = []

1.00

0.95

0.90

0.85

0.80 0.80 0.75

0.70

0.65

r_squared = []

if feature == 0:

elif feature == 1:

np.random.seed(random_state)

for feature in range(num_features):

R^2 will only go up as additional features are added, even if those features provide the model no new predictive power. See the

def generate_random_data(num_rows, num_features, random_state=None)

data = np.random.uniform(size=num_rows)

data = np.hstack((data.reshape(-1,1),

'''returns 2D numpy array based on user-defined inputs with ran-

data = np.hstack((data, np.random.uniform(size=num_rows)

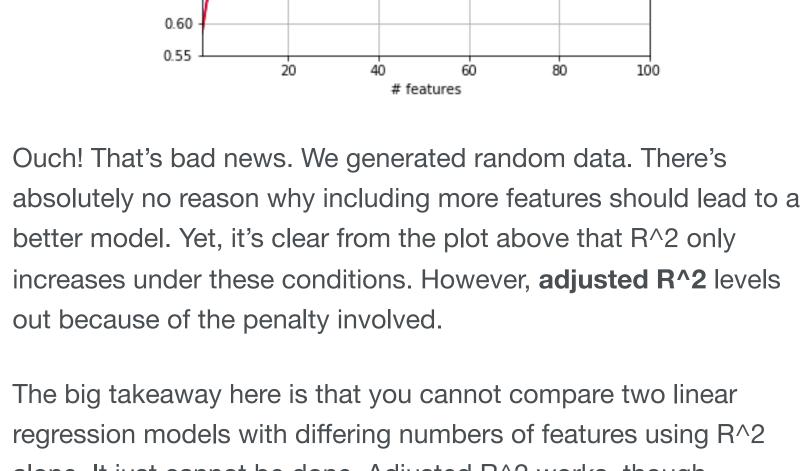
np.random.uniform(size=num_rows).resh

R2 adj

Machine Learning

Next

for features in range(min_features, max_features+1): X = generate_random_data(num_rows=200, num_features=features, re



train/test split and cross-validation.

Data Science

Previous

Categories:

Python

SHARE ON G+ Google+ f Facebook in LinkedIn **W** Twitter