Predicting House Prices with Linear Regression | Machine Learning from Scratch (Part II)

Venelin
Valkov
Apr 1 · 7 min read

> *Predicting sale prices for houses, even stranger ones. And what's up with that basement?*

> *TL;DR Use a test-driven approach to build a Linear Regression model using Python from scratch. You will use your trained model to predict house sale prices and extend it to a multivariate Linear Regression.*

I know that you've always dreamed of dominating the housing market. Until now, that was impossible. But with this limited offer you can… got a bit sidetracked there.

Let's start building our model with Python, but this time we will use it on a more realistic dataset.

Complete source code notebook (Google Colaboratory):

LinearRegression

colab.research.google.com

The Data

Our data comes from a Kaggle competition named “House Prices: Advanced Regression Techniques”. It contains *1460* training data points and 80 features that might help us predict the selling price of a house.

Load the data

Let’s load the Kaggle dataset into a Pandas data frame:

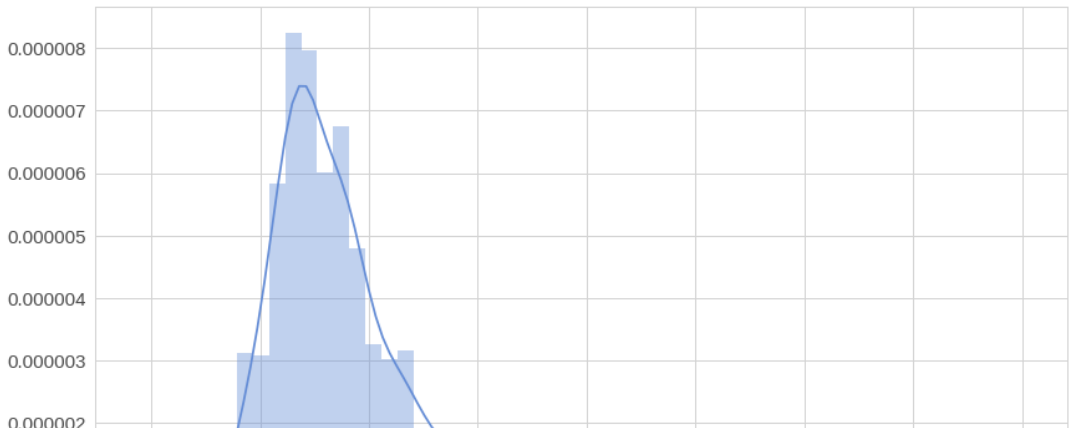
```
df_train = pd.read_csv('house_prices_train.csv')
```

linr-load-data.py hosted with ❤ by **GitHub** **view raw**

Exploration—getting a feel for our data

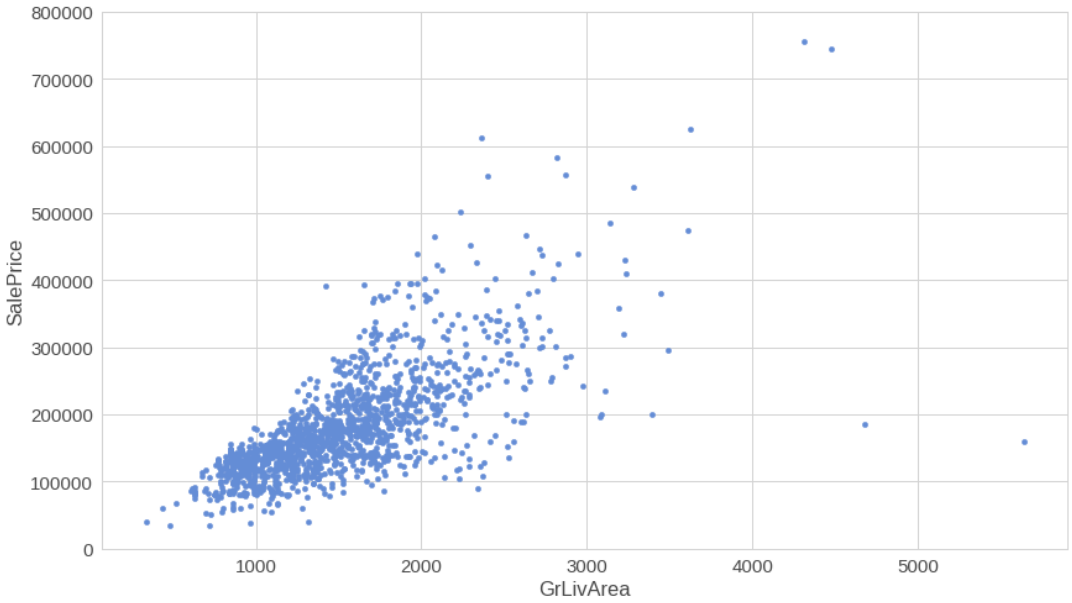
We’re going to predict the `SalePrice` column ($ USD), let’s start with it:

```
count     1460.000000
mean    180921.195890
std      79442.502883
min      34900.000000
25%     129975.000000
50%     163000.000000
75%     214000.000000
max     755000.000000
Name: SalePrice, dtype: float64
```



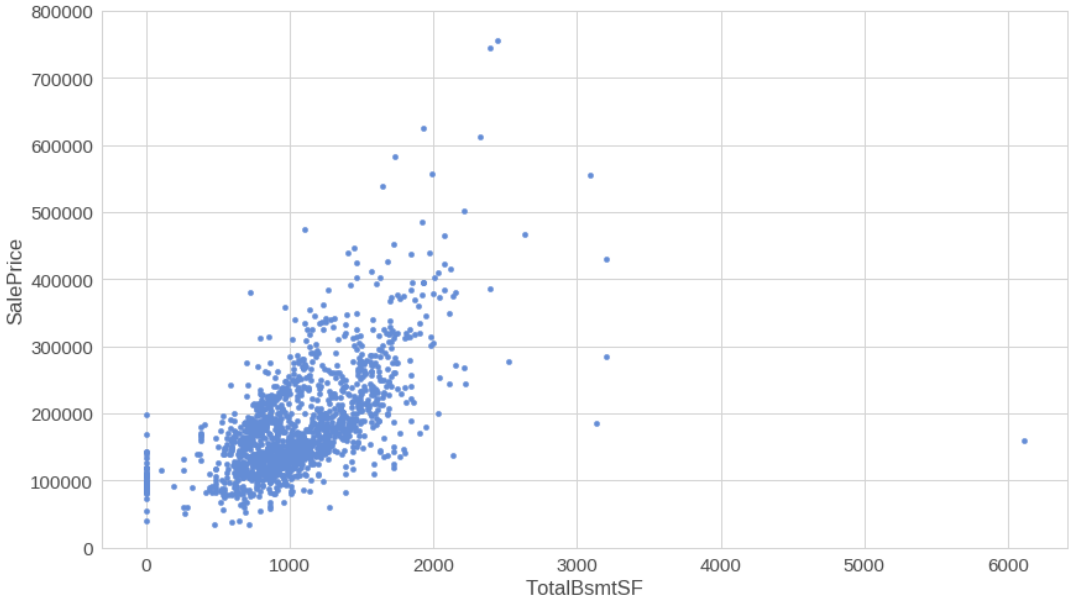
Most of the density lies between 100k and 250k, but there appears to be a lot of outliers on the pricier side.

Next, let's have a look at the greater living area (square feet) against the sale price:



You might've expected that larger living area should mean a higher price. This chart shows you're generally correct. But what are those 2–3 "cheap" houses offering huge living area?

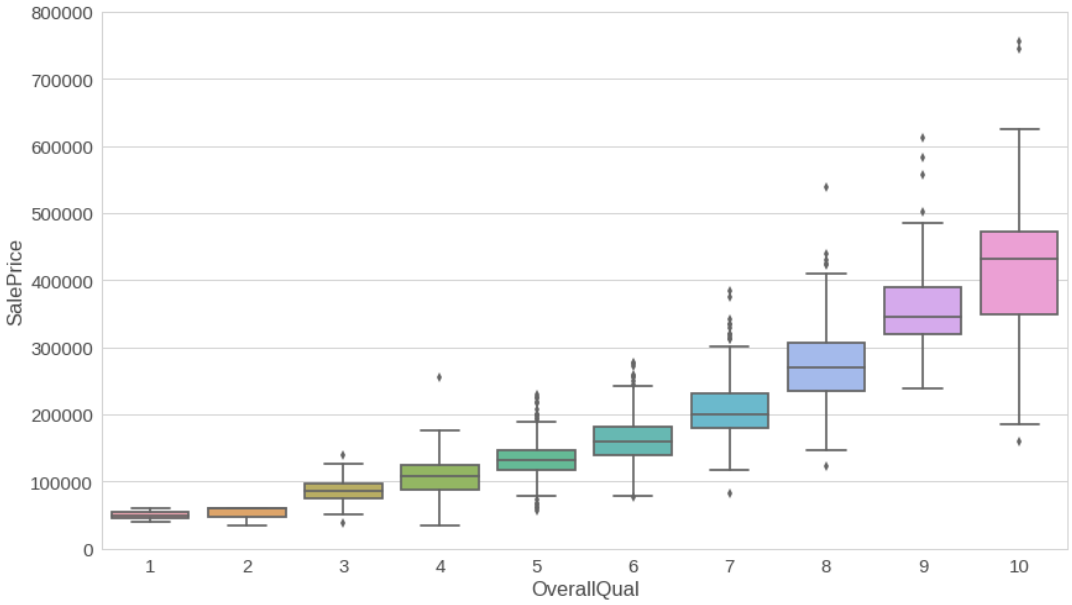
One column you might not think about exploring is the "TotalBsmtSF"—Total square feet of the basement area, but let's do it anyway:



Intriguing, isn't it? The basement area seems like it might have a lot of predictive power

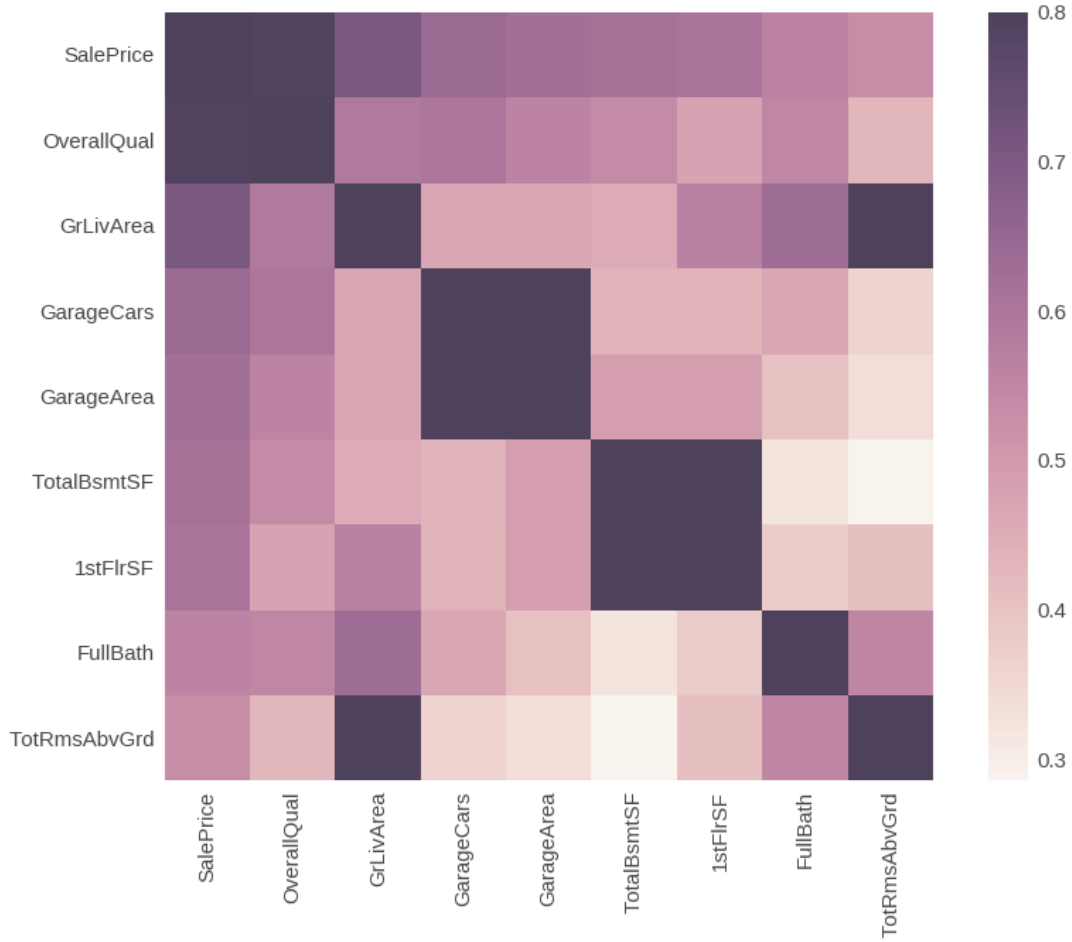
for our model.

Ok, last one. Let's look at "OverallQual"—overall material and finish quality. Of course, this one seems like a much more subjective feature, so it might provide a bit different perspective on the sale price.



Everything seems fine for this one, except that when you look to the right things start getting much more nuanced. Will that "confuse" our model?

Let's have a more general view on the *top 8* correlated features with the sale price:



Surprised? All the features we discussed so far appear to be present. Its almost like we knew them from the start…

Do we have missing data?

We still haven't discussed ways to "handle" missing data, so we'll handle them like a boss—just not use those features:

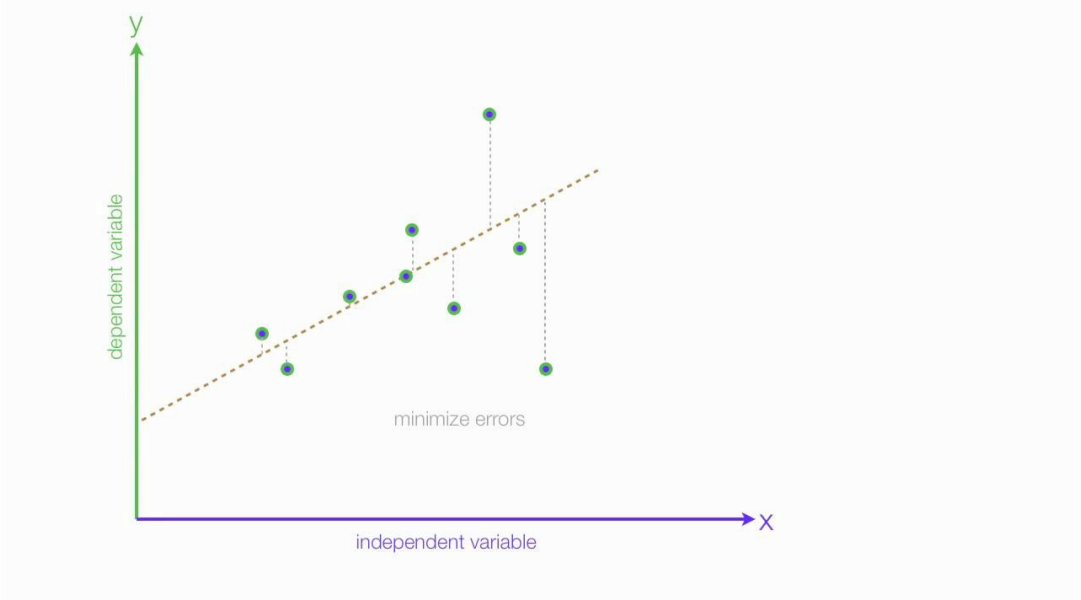
	Row count	Percentage
PoolQC	1453	0.995205
MiscFeature	1406	0.963014
Alley	1369	0.937671
Fence	1179	0.807534
FireplaceQu	690	0.472603
LotFrontage	259	0.177397
GarageCond	81	0.055479
GarageType	81	0.055479
GarageYrBlt	81	0.055479
GarageFinish	81	0.055479
GarageQual	81	0.055479
BsmtExposure	38	0.026027
BsmtFinType2	38	0.026027
BsmtFinType1	37	0.025342
BsmtCond	37	0.025342

Yes, we're not going to use any of those.

Predicting the sale price

Now that we have some feel of the data we're playing with we can start our plan of attack—how to predict the sale price for a given house?

Using Linear Regression



source: http://mybooksucks.com

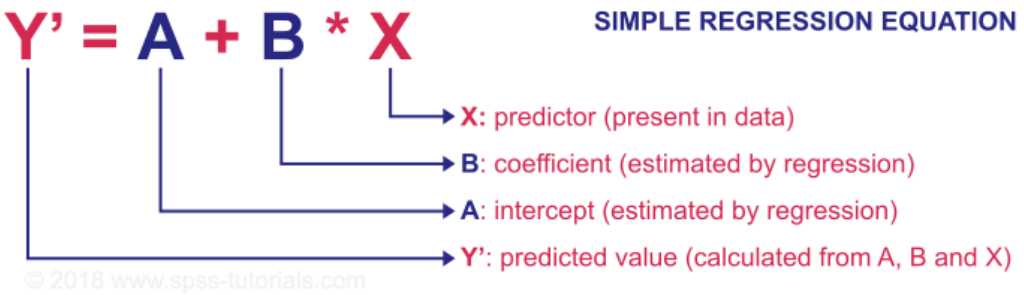
Linear regression models assume that the relationship between a dependent continuous variable Y and one or more explanatory (independent) variables X is linear

(that is, a straight line). It's used to predict values within a continuous range (e.g. sales, price) rather than trying to classify them into categories (e.g. cat, dog). Linear regression models can be divided into two main types:

Simple Linear Regression

Simple linear regression uses a traditional slope-intercept form, where *a* and *b* are the coefficients that we try to "learn" and produce the most accurate predictions. *X* represents our input data and *Y* is our prediction.

$$Y = bX + a$$



source: https://spss-tutorials.com

Multivariable Regression

A more complex, multi-variable linear equation might look like this, where *w* represents the coefficients or weights, our model will try to learn.

$$Y(x_1, x_2, x_3) = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_0$$

The variables `x_1, x_2, x_3` represent the attributes or distinct pieces of information, we have about each observation.

Loss function

Given our Simple Linear Regression equation:

$$Y = bX + a$$

We can use the following cost function to find the coefficients/parameters for our model:

Mean Squared Error (MSE) Cost Function

The MSE is defined as:

$$MSE = J(W) = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - h_w(x^{(i)}))^2$$

where

$$h_w(x) = g(w^t x)$$

The MSE measures how much the average model predictions vary from the correct values. The number is higher when the model is performing “bad” on our training data.

The first derivative of MSE is given by:

$$MSE' = J'(W) = \frac{2}{m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})$$

One Half Mean Squared Error (OHMSE)

We will apply a small modification to the MSE—multiply by *1/2* so when we take the derivative, the *2s* cancel out:

$$OHMSE = J(W) = \frac{1}{2m}\sum_{i=1}^{m}(y^{(i)} - h_w(x^{(i)}))^2$$

The first derivative of OHMSE is given by:

$$OHMSE' = J'(W) = \frac{1}{m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})$$

Let’s implement it in Python (yes, we’re going TDD style!)

```
class TestLoss(unittest.TestCase):

  def test_zero_h_zero_y(self):
    self.assertAlmostEqual(loss(h=np.array([0]), y=np.array([0])), 0)

  def test_one_h_zero_y(self):
    self.assertAlmostEqual(loss(h=np.array([1]), y=np.array([0])), 0.5)

  def test_two_h_zero_y(self):
    self.assertAlmostEqual(loss(h=np.array([2]), y=np.array([0])), 2)

```

Now that we have the tests ready, we can implement the loss function:

```
def loss(h, y):
  sq_error = (h - y)**2
  n = len(y)
  return 1.0 / (2*n) * sq_error.sum()
```

ohmse.py hosted with ❤ by **GitHub** **view raw**

```
run_tests()
```

time for the results:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.007s

OK
```

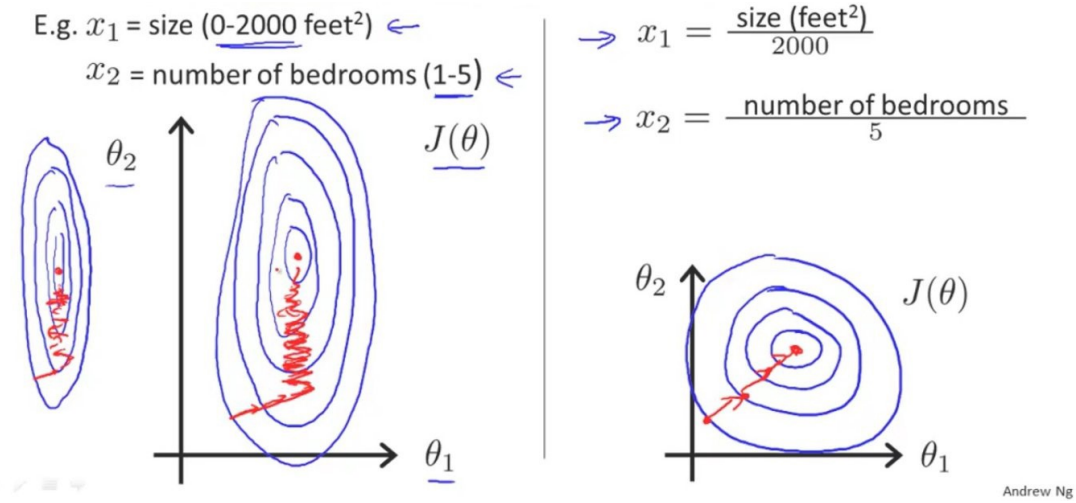
Data preprocessing

We will do a little preprocessing to our data using the following formula (standardization):

$$x' = \frac{x - \mu}{\sigma}$$

where μ is the population mean and σ is the standard deviation.

But why? Why would we want to do that? The following chart might help you out:



source: Andrew Ng

What the doodle shows us that our old friend—the gradient descent algorithm, might converge (find good parameters) faster when our training data is scaled. Shall we?

```
x = df_train['GrLivArea']
y = df_train['SalePrice']

x = (x - x.mean()) / x.std()
x = np.c_[np.ones(x.shape[0]), x]
```

lin-reg-scaling.py hosted with ❤ by **GitHub** **view raw**

We will only use the `greater living area` feature for our first model.

We will only use the `greater living area` feature for our first model.

Implementing Linear Regression

First, our tests:

```
class TestLinearRegression(unittest.TestCase):

    def test_find_coefficients(self):
      clf = LinearRegression()
      clf.fit(x, y, n_iter=2000, lr=0.01)
      np.testing.assert_array_almost_equal(clf._W, np.array([180921.19555322,  56294.9019992
```

linear-regression-test.py hosted with ❤ by GitHub view raw

Without further ado, the simple linear regression implementation:

```
class LinearRegression:

  def predict(self, X):
    return np.dot(X, self._W)

  def _gradient_descent_step(self, X, targets, lr):

    predictions = self.predict(X)

    error = predictions - targets
    gradient = np.dot(X.T,  error) / len(X)
```

You might find our Linear Regression implementation simpler compared to the one presented for the Logistic Regression. Note that the use of the Gradient Descent algorithm is pretty much the same. Interesting, isn't it? A single algorithm can build two different types of models. Would we be able to use it for more?

```
run_tests()
```

```
.....
---------------------------------------------------------------------- Ran 6 tests in 1.094s

OK
```

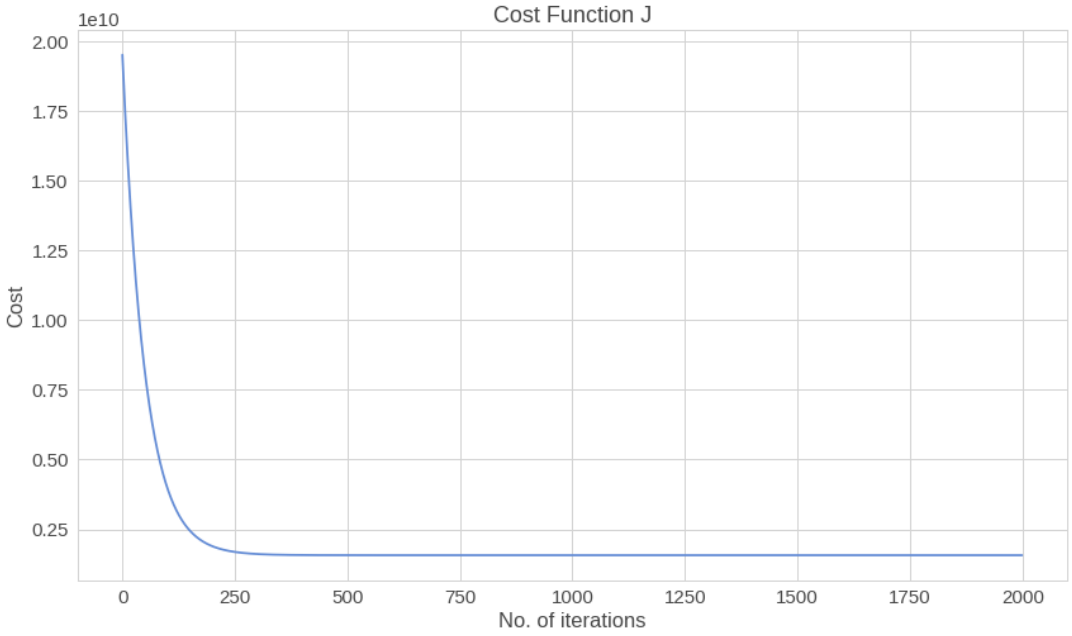
Predicting the sale price with our first model

Let's use our freshly created model to begin our housing market domination:

```
clf = LinearRegression()
clf.fit(x, y, n_iter=2000, lr=0.01)
```

linear-regression-train.py hosted with ❤ by GitHub view raw

So how did the training go?



Our cost, at the last iteration, has a value of:

```
1569921604.8332634
```

Can we do better?

Multivariable Linear Regression

Let's use more of the available data to build a *Multivariable Linear Regression* model and see whether or not that will improve our OHMSE error. Let's not forget that scaling too:

```
x = df_train[['OverallQual', 'GrLivArea', 'GarageCars']]

x = (x - x.mean()) / x.std()
x = np.c_[np.ones(x.shape[0]), x]
```

multivariable-lin-reg-data.py hosted with ❤ by **GitHub** **view raw**

Implementing Multivariable Linear Regression

> *This space is intentionally left (kinda) blank*

Using Multivariable Linear Regression

Now that the implementation of our new model is done, we can use it. Done!?

You see, young padawan, the magical world of software development has those mythical creates called ***abstractions***. While it might be extraordinarily hard to get them right, they might reduce the complexity of the code you write by (pretty much) a ton.

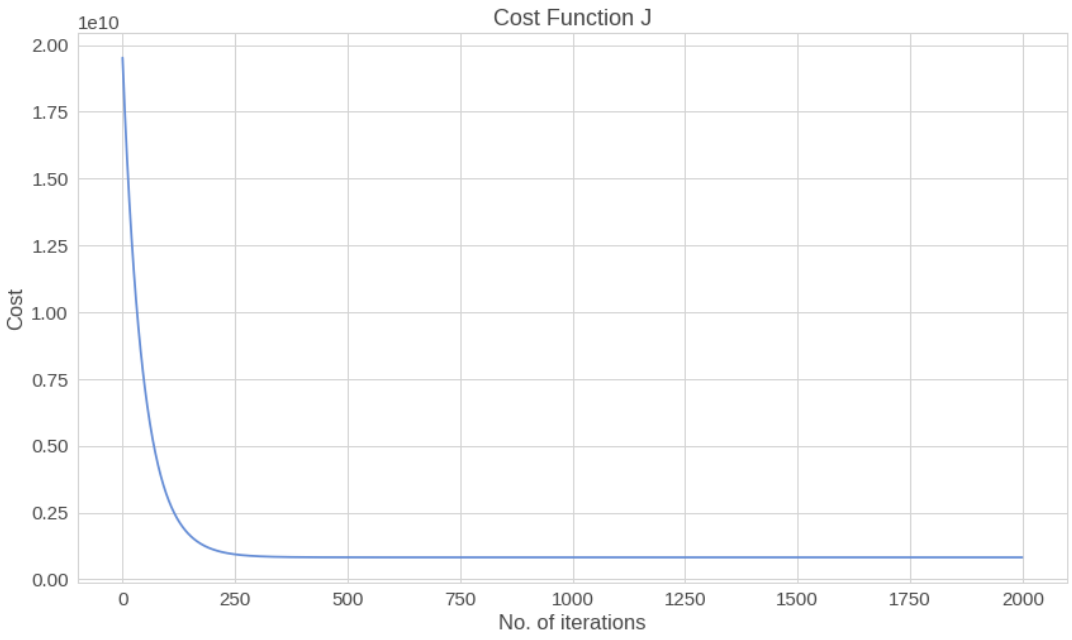
You just used one such abstraction—called ***vectorization***. Essentially, that allowed us to build a *Multivariable Linear Regression* model without the need to loop over all features in our dataset. Neat, right?

Our client interface stays the same, too:

```
clf = LinearRegression()
clf.fit(x, y, n_iter=2000, lr=0.01)
```

linear-regression-train.py hosted with ❤ by **GitHub** **view raw**

And the results:



822817042.8437098

The loss at the last iteration is nearly 2 times smaller. Does this mean that our model is better now?

You can find complete source code and run the code in your browser here:

LinearRegression

colab.research.google.com

Conclusion

Nice! You just implemented a Linear Regression model and not the simple/crappy kind.

One thing you might want to try is to predict house sale prices on the testing dataset from Kaggle. Is this model any good on it?

In the next part, you're going to implement a Decision Tree model from scratch!

Machine Learning | Data Science | Artificial Intelligence | Linear Regression | Python

1
clap

Venelin Valkov
Adventures in Artificial Intelligence
https://www.mypoli.fun/

Follow