6CCS3ML1 (Machine Learning)

Practical 2: Regression and gradient descent

(Version 1.1)

1 Overview

This practical builds on Lecture 2. In that lecture we discussed regression, and the use of gradient descent to learn linear models. In this practical we will use these ideas to create a regression model. To do this, you will (again) make use of `scikit-learn`, a Python library that includes several different kinds of learner.

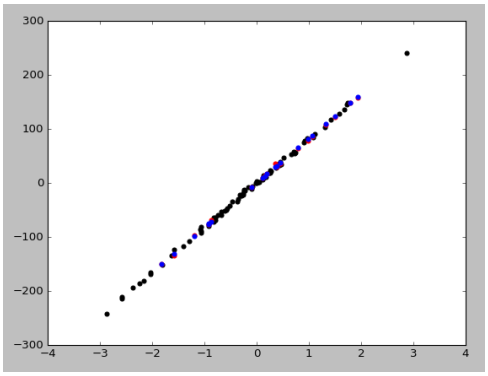
2 Using linear regression

We will start by using `scikit-learn` to perform regression. (In fact we will use it to both create and solve a linear regression problem.)

Download

`regression.py`

from KEATS and run it. You should get output that looks like this:



There are three things in this image. (You may have to expand the window to see them clearly.) First, there are black dots. These are training data. As you can see, there is a linear relationship between them. Second, there are red dots. These are test data drawn from the same distribution as the training data. Finally, there are blue dots. These are the result of feeding the test data through a regression model that was learnt on the training data.

Does it look as though the regression model did a good job of learning? Why?

Take a look at the code. This is a bit different to what you have seen before. Rather than start by importing a `scikit-learn` dataset, it starts by using `scikit-learn` to generate data:

```
X, y = make_regression(n_samples=100, n_features=1, noise = 3)
```

This creates a set of 100 data points, each of which has 1 feature, in an approximately linear pattern. The "approximately" is controlled by the parameter `noise`, which feeds in Gaussian noise. The larger the value of `noise`, the more the points are scattered.

The lines:

```
regr = linear_model.LinearRegression()
regr.fit(X_train, y_train)
```

creates a regression model called `regr`, and uses the data `X_train` and `y_train` to learn from. As with the models in the previous practical, the function that does the learning is `fit( )`

The next section of the code evaluates the model that was just learnt. This:

```
print("Mean squared error: %.2f" % np.mean((regr.predict(X_test) - y_test) ** 2))
```

computes the mean squared error on the test data. That means it takes the values in `X_test`, and runs them through the regression model to make predictions about the corresponding `y` value should be. These predictions are the blue dots. It then takes the real corresponding `y` value, the one in `y_test` and calculates the difference between, and squares it. The average of these values is the mean squared error. So the smaller, the better.

This:

```
print('Variance score: %.2f' % regr.score(X_test, y_test))
```

computes another metric, the *variance score*. The key thing about this is that the maximum (and best) value is 1.0. This means that, as far as the test data is concerned, the model is a perfect fit.

Finally, these lines:

```
plt.scatter(X_train, y_train, color="black")
plt.scatter(X_test, y_test, color="red")
plt.scatter(X_test, regr.predict(X_test), color="blue")
```

generate the output. In turn they produce three scatter plots. The first is the training data. The second is the test data. The third is the result of running the regression model on `X_test`. It thus shows the regression model in action. The error is the distance between each red dot and the corresponding blue dot.

Now do the following:

1. Experiment with noise parameter.

   As you vary it, look at the effect on the mean squared error and the variance.

3 Doing gradient descent

The previous section had you use the built-in function in `scikit-learn` to do regression. You will now do this for yourself. This means implementing simple gradient descent. The following items are not steps, but are things to think about as you develop the regression model. Read through them all before you begin.

This is the process described in the slides from Lecture 2.

- Start from `regression.py`, and to make things easy, set the `noise` in the data generator to a low value.
- A simple regression model takes the form of:

$$y = w_0 + w_1.x$$

  Since we have just one feature, x is just one value.
- Training the regression model involves taking a value x_i from `X_train`, and using the regression model to make a prediction about its value:

$$p_i = w_0 + w_1.x_i$$

- The difference between this prediction and the corresponding value y_i in `y_train` is the error:

$$error_i = y_i - p_i$$

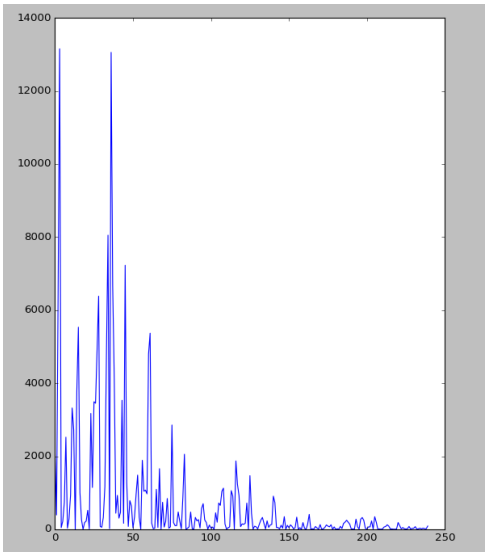
- We use the error to adjust the value of both w_0 and w_1:

$$w_0 \leftarrow w_0 + \alpha.error_i$$
$$w_1 \leftarrow w_1 + \alpha.error_i.x_i$$

- We repeat this until we think we have a good model. That may mean using each element in the training set more than once.
- Note that the method I have sketched needs a *learning rate*, α. This should be less than 1. Typically we set α to be small. That means the learning will take longer to converge, but it is more likely to converge. That is usually a good tradeoff.

What I have described above is an implementation of stochastic gradient descent (see the slides from Lecture 2).

To see how learning is proceeding, plot the error after each prediction. You should see something like this:



This is a form of learning curve, like we saw in the lectures. Since this curve measures error, it is the dual of the curves we saw in the lecture. Note how noisy it is — over time the error decreases, but it still makes quite big errors well into the learning.

What you should do is:

1. Implement gradient descent as described above.
2. Plot the error/learning curve.
3. Compute the mean squared error for your model on your test data.
4. Modify your model (especially learning rate and the number of times the training data is used) until it performs as well as the `scikit-learn` built in method.
5. Vary the noise in the dataset and see how well your learning method compares with the `scikit-learn` method.

4 Multivariate gradient descent

The function that we are using to generate the dataset will produce data with many features. For example:

```
X, y = make_regression(n_samples=100, n_features=3, noise = 3)
```

will create a dataset with 3 features (that is three values of x for every value of y). For such a dataset, a suitable regression model is:

$$y = w_0 + w_1.x_1 + w_2.x_2 + w_3.x_3$$

We can learn this exactly was before, but we have to update four weights at every iteration rather than two.

You should:

1. Extend your gradient descent program to handle multiple features.
2. Again, test this against the performance of the `scikit-learn` builtin function using means squared error.

5 More

If you want to try other things:

1. So far we have been using stochastic gradient descent to train the regression model. Extend your code to do batch gradient descent as well.

   Which works better?

2. Look at the Boston Housing dataset from `scikit-learn`. Import this using:

```
from sklearn.datasets import load_boston
boston = load_boston()
```

This is not a very linear dataset, but if you pull out feature number 5:

```
X_train, X_test, y_train, y_test = train_test_split(
  boston.data[:, np.newaxis, 5], boston.target, test_size=0.2, random_state=0)
```

you can find something that is reasonably linear (I get a mean squared error of around 46 and a variance score of 0.42).

3. Now do a multivariate regression on the Boston Housing data. Can you find a model that fits better?

4. Finally, go back to the decision tree model that you built for the Wisconsin Breast Cancer dataset in the last practical. Turn that into an ensemble of decision trees.

   Note that you will likely find that an ensemble which is just a bunch of decision trees built from subsets of the data doesn't perform *that* well.[1] That is because the trees you get are not different enough (so their outputs are correlated and not independent). This is why the random forest approach uses bagging and random subspace method.

6 Version list

- Version 1.0, January 18th 2020.
- Version 1.1, January 11th 2021.

[1] I found that splitting the training data into 10 parts and learning 10 trees from it gave an ensemble that outperformed most of the individual trees, but not all (so it was kind of averaging their performance), but often beat a single tree learnt from all the training data.