COSC 2673 Machine Learning

Lab 03

Objective

- Continue to familiarise with Python
- Familiarise with Anaconda and other Python Machine Learning tools
- Perform regression
- Learn how to perform basic evaluation (we learn more sophisticated approaches in coming weeks)

Introduction

In this lab, we will apply regression to the datasets we explored last week, namely the Boston house price and Bike Hire Count datasets. We will study how to do univariate and multi-variate regression, and how to perform evaluation to compare the performance of the different regression models we build.

Datasets

In this lab, we will be using the Boston house price dataset and Bike Hire count datasets.

Regression Tool Description

This lab, we will continue to use the libraries we studied, including pandas, Matplotlib and Numpy. In addition, this week we additionally use another great machine learning library called Scikit-learn, which has implementations of many of the algorithms we will learn. Note, these aren't the only libraries that are useful for machine learning, as with Python, there are multiple ways to achieve the same aims, so we encourage you to try these and others and see what works best for you. But for the lab solutions, we will implement in terms of the ones we work on in labs.

Uni-variate Regression

We will first study how to do uni-variate regression.

Open up Anaconda and create a new Jupiter Notebook session (ask your demonstrator if you have issues). Download the datasets into the working directory of your session, to make it easier to find and load the datasets.

COSC 2673 Machine Learning

Lab 03

Once you have the session and data downloaded, first import the necessary packages (we will examine what each of them do as we go along the lab). Type the following into the first cell of the Jupiter session (without the $, which denotes the prompt):

```
$ import pandas as pd
$ import matplotlib.pyplot as plt
$ import numpy as np
$ import sklearn as sk
```

If you unsure or forgotten about some of the Python syntax, please look up what 'import' does (it essentially loads the package and associated classes and functions into your session so you can use them).

Next, we want to load the dataset and print it out to check it has been loaded correct:

```
$ bostonHouseFrame = pd.read_csv('../BostonHousingPrice/housing.data.csv',
delim_whitespace=True)
$ print(bostonHouseFrame)
```

This is similar to last week, where we are using pandas (which has an alias 'pd' from the import statement) to do so. bostonHouseFrame is a data frame, which if you recall contains both the column headings and actual tabular data.

If you recall from last lab, we found that possibly the 'RM' (number of rooms) and 'LSTAT' (unsure) variables seems to have a linear relationship with the house price ('MEDV'). Hence, we will try these variables as the 'X', independent variable to predict the house price, the dependent variable. However, we first need to create our X and Y variables:

```
$ uniRmX = bostonHouseFrame [['RM']]
$ Y = bostonHouseFrame[['MEDV']]
```

Creates an X variable that is just based on the 'RM' column, and a Y variable of 'MEDV'.

Next, we want to create some data to train the model, and some other data to evaluate how good the model is. We may be tempted to use 100% of the data for training, then select a subset for evaluation (say 20% of the data). However, we will find out later that this isn't a good idea generally, as the data we use to test is the same we use to train, and likely to cause overfitting and what is called bias. We will discuss this more in later week, but for now just note that it isn't a good idea to do so.

Instead, we will split the data into a training set, which we use to fit the model/hypothesis, and a testing set, which we will use to evaluate the performance of the fitted model/hypothesis. There should be no overlap between the two sets. How we achieve this is typically randomly split the data, say 80% for training and 20% for testing. We can do this

ourselves, but like many machine learning functionality these days, they are already implemented, and of course, they are available in the libraries we have imported.

```
$ # create testing and training data for RM variable
$ from sklearn.model_selection import train_test_split

$ trainX, testX, trainY, testY = train_test_split(np.array(uniRmX), np.array(Y),
test_size=0.2)
$ print(trainX.shape)
$ print(testX.shape)
$ print(trainY.shape)
$ print(testY.shape)
```

The function that does the work is 'train_test_split()', part of sklearn.model_selection package. It essentially does what we desire, with the first two arguments to it are the X and Y variable datasets respectively (they must be of the same number of rows/instances), and a test_size parameter which specifies how big the test dataset is (in this case, 20% of the data, which is a typical setting for this). This function returns 'trainX' (training data of the X variable), 'testX' (testing data of X), 'trainY' (training data of Y) and 'testY' (testing data of Y). The last four statements just prints out the size of the resulting training and testing datasets to show you that the training (test) datasets contains the same number of rows. We will use trainX and trainY to train the linear regression model, then textX and testY for testing and evaluation.

We are now ready to construct the linear regression model/hypothesis and fit the theta parameters (recall θ_1 and θ_0).

```
$ from sklearn import linear_model
$ linReg = linear_model.LinearRegression()
```

This constructs the linear regression model object, assigned to variable 'linReg'. We then fit the training data to the model:

```
$ linReg.fit(trainX, trainY)
```

linReg.fit() fits the X and Y training data, and optimises the parameters to minimise the loss function. It might not exactly use gradient descent, but the ideas are similar and as stated in lectures, gradient descent as a general optimisation is the crux of many optimisation and parameter fitting algorithms.

Not too difficult right?

Lets have a look at what the parameters look like:

```
$ print(linReg.intercept_)
```

$ print(linReg.coef_)

linReg.intercept_ is the y-intercept, or essentially the θ_0 parameter. linReg.coef_ is slope of the univariate linear regression line, or the θ_1 variable.

With a model/hypothesis, we can now do prediction! We use the testing data for that:

$ predYRm = linReg.predict(testX)

Which predicts the Y value for each input X value in testX testing dataset. The predictions are stored in 'predYRm'.

Okay, we now have predictions, but how did we go? What we want to do is to compare the predicted value from our model (given testX) with the actual Y values (testY). We can estimate the error (using the mean squared error loss function we been discussing in lectures for fitting the parameters), as follows:

$ from sklearn.metrics import mean_squared_error
$ print('Mean squared error ', mean_squared_error(testY, prediction))

What is the mean squared error value you obtained?

In addition, it is useful for regression to plot the testing data against the model/hypothesis, which is a line in our univariable linear regression. Type in the following:

$ plt.scatter(testX, testY, color='black')
$ plt.plot(testX, prediction, color='blue', linewidth=3)

This uses our reliable plotting package matplotlib and pyplot to first, produce a scatterplot of our testing data (X,Y) pairs, then draw a blue line for our model/hypothesis. What information does the plot visualise? Discuss with your demonstrator and lab mates.

Another interesting visualisation and based on the same information is the residual plot, which shows what is the difference in predicted and actual values, across different X values, for the testing data;

$ plt.scatter(testX, linReg.predict(testX) - testY, c='g', s=40)
$ plt.hlines(y=0, xmin=3.5, xmax=9)

If the model is perfect, then for each testing data then there is 0 residual (or 0 error between predicted versus actual), which is represented by the horizontal line in the diagram. However, if the model under-estimates the actual value, it will be below this horizontal line, and if over-estimates, than above. This plot can quickly give you a sense of where the errors are occurring and whether there are outlier points that are causing large errors (we will examine this later in the course and ways to deal with this).

COSC 2673 Machine Learning

Lab 03

Exercise

We have fitted a linear regression model/hypothesis for the ‘Rm’ variable. We also saw that the ‘LSTAT’ variable seem to have a linear relationship with our target variable ‘MEDV’. Repeat above analysis using the ‘LSTAT’ variable, and comment on which X variable you think is performing better. LSTAT variable has a negative (inverse) linear relationship with ‘MEDV’, where when LSTAT increases MEDV decreases – do you think it matters? Discuss with your demonstrator.

Exercise

So far, we have only used one X variable to predict MEDV. What if we used all the variables? To create such data, we first do the following:

```
$ X = bostonHouseFrame.drop('MEDV', axis=1)
$ Y = bostonHouseFrame[['MEDV']]
```

The Y variable is the same, but the X variable is interesting. We use the drop() method of data frames, which essentially drops a column from it – in our case, we drop the ‘MEDV’ column. X is now a data frame without the MEDV column, print it out the check.

Now repeat the same analysis, noting that train_test_split(), linReg.fit() and linReg.predict() can work with multi-variate linear regression (i.e. where we have multiple X variables) as well. Make sure you print out the intercept and coefficients, print(linReg.intercept_) print(linReg.coef_), to see what are the fits and do the mean squared error

Extension Exercise

As further practice, repeat the same exercise for the bike share data.