Practical 2: Regression and Classification of Numeric Data

1 Overview

The aim of these lab exercises is to give you some hands-on experience with regression and classification of numeric data. There are two parts: **linear regression**, and **classification with perceptrons**. For these exercises, you will work with a data set provided and you will also learn how to generate *synthetic* data sets—i.e., sets of randomly generated numbers that will help you perform the tasks.

2 Getting Started

You will need do install **Python** and **scikit-learn** in order to complete these lab exercises (as well as the rest of the lab exercises and the Coursework for this module).

1. I recommend installing the **Anaconda** distribution of **Python v3.8** into the `/home/` directory of your NMS computer account or your laptop computer—whichever you plan to use during the Practicals for this module. You can download it from here:

   `https://www.anaconda.com/download/`

   If you have trouble completing this, please ask a TA for help.

   NOTE: If you took 7CCSMCMP in term 1, then you might have already done this.

2. You should already have the **scikit-learn** package installed from when you installed Anaconda. However, if for some reason you do not, then you can install it using:

```
unix-prompt$ conda install scikit-learn
```

3 Prepare Your Data

The first step for any data mining task is to gather and prepare your data. In this lab, we will work with two types of data: (a) data that I have provided for you; and (b) so-called **synthetic** data. The latter type of data is useful for you to employ when learning new techniques, because you can specify the characteristics of the data you would like to have and then use a tool to generate random instances of that type of data. Complete the two sections below in order to have your data ready for the exercises that follow.

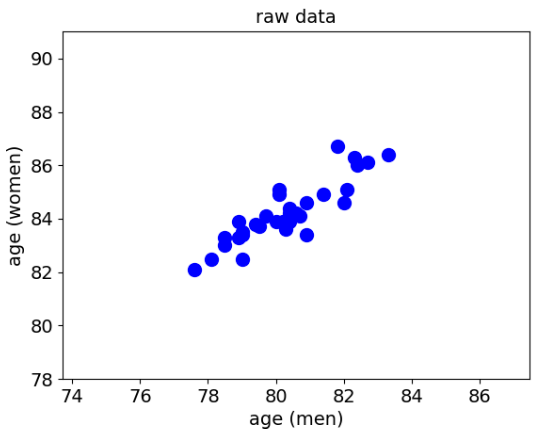
3.1 Download data

1. On the KEATS page for this module (7CCSMDM1), under **[Week 2]**, find and download the **.zip** archive file containing the data sets which will be used in this practical.

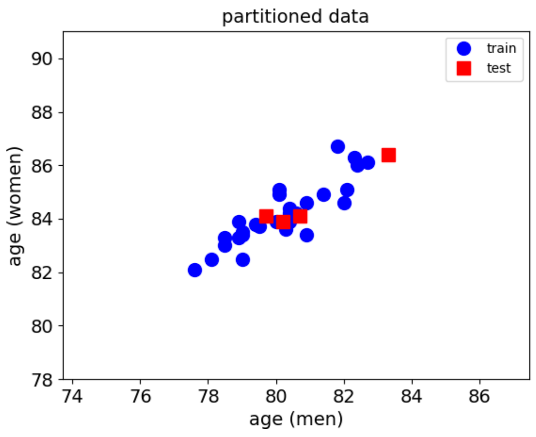
2. Unzip the archive.

3. Inside the archive, you will find the data file called: **london-borough-profiles-jan2018.csv**. This is a mixed data set with 85 different variables of various types. For this exercise, use columns 70 and 71 (assume we start numbering columns 0), which are the values of *Male life expectancy, (2012-14)* and *Female life expectancy, (2012-14)* in different boroughs in London.

4. Write some Python code to read in the data and plot it to see what it looks like. Try setting x to Male life expectancy and y to Female life expectancy. Your plot should look something like Figure 3.1a.



(a) raw data



(b) partitioned data (discussed in part 3.2)

Figure 1: London Borough Profiles data set (2018)

3.2 Partition the data

1. As we discussed in class this week, it is a bad idea to use the same data set for both training and evaluating. The reason is because your evaluation score will only indicate how well your model represents your training data and not how well it represents "unseen" data. Thus, you want to divide your data sets into two portions or *"partitions"*. For example, it is common to use 90% for **training** and 10% for **testing**. You could use any proportion you want (as long as they add up to 100%).

   You can partition data using the **slice** function in Python. You can also do this by using the **scikit-learn** function train_test_split (), as shown in Figure 2.

```
from sklearn import model_selection
# partition the data
x_train, x_test, y_train, y_test = model_selection.train_test_split( x, y,
    test_size=0.10 )
```

Figure 2: Example code for partitioning the data into training and test sets

→ Modify your code from part 3.1 to partition the data into training and test sets.

2. Now plot your data again, showing the two partitions. Your plot should look something like Figure 3.1b.

3. Note that we also talked this week about **cross-validation**, where you partition the data multiple times. For this week, we'll only partition once. We will employ cross-validation in later lab exercises, so you'll get a chance to try that technique too.

3.3 Generate a synthetic dataset

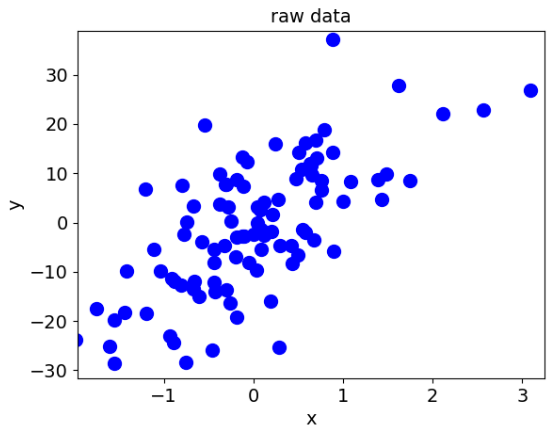
Sometimes, it is useful to be able to generate a *synthetic* data set where you specify the number and data type of attributes and you can specify some properties of each attribute (such as a valid numeric range). For this part of the lab, you will generate such a data set.

1. Randomly generate a synthetic dataset of 100 data points (x, y), where the target variables, y, are drawn from a linear model $y = p\,x + q$ with some randomly selected p and pre-defined noise factor q. We do that using the *scikit-learn* function `datasets.make_regression()`, as shown in Figure 3.

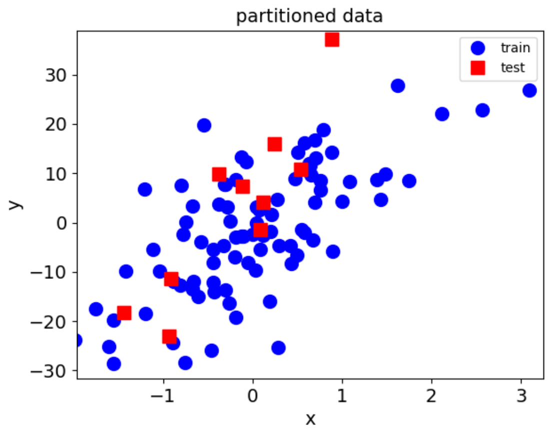
2. As with the downloaded data set, it is always good to plot the data before trying to analyse it. Your plots should look something like Figure 4.

```
#--
# generate some random data for regression
#--
from sklearn import datasets
x, y, p = datasets.make_regression( n_samples=100, n_features=1, n_informative=1,
    noise=10, coef=True )
```

Figure 3: Example code for generating synthetic data for linear regression. The function parameters are: `n_features` = defines how many attributes to generate for each instance; `n_informative` = how many of those are used to infer the target variables; `noise` = the noise factor q; `n_samples` = the number of instances to generate; and `coef` = a flag indicating whether to return the p coefficients used to generate the target values y. The function returns: x, the generated 2D array of features, of size *n_samples* × *n_features*; y, an array of target values, also of size *n_samples*; and p, an array of parameters, also of size *n_samples*.



(a) raw data



(b) partitioned data

Figure 4: Example synthetic data set for linear regression

4 Linear Regression

As we discussed in class, **Linear Regression** is a useful method for investigating relationships between a target variable y and set of variables (or attributes) χ. In this lab, you will do this in two ways. First, you'll write your own code to perform **gradient descent**, which we discussed in class. This will help you understand how the algorithm works. You should find it useful to plot intermediate solutions, as the method iterates, so you can quite literally watch your solution converge.

Then, you'll use the **scikit-learn** version of linear regression. This library provides a number of built-in options, which you could write yourself by extending the code you just wrote. Or, now that you understand the basics of how the method works (because you just wrote the fundamentals yourself), you could take advantage of the code that someone else wrote and tested and helpfully posted on the internet...

4.1 Solving Linear Regression with Gradient Descent

1. Write code in Python to perform **gradient descent** in order to fit the parameters ω of a linear model. Recall from the lecture that the gradient descent algorithm works as follows: on the first step, the parameters (ω) are initialised with some random values (or set to 0s); and then they are gradually updated in the direction that minimises the objective function, using the following update rule:

$$w_i \leftarrow w_i + \alpha(y_j - \hat{y}_j)\ x_{j,i} \tag{1}$$

$\rightarrow$ Complete the function gradient_descent_2 (), shown in Figure 5, which implements iterative updates of the parameters ω according to Equation 1.

2. Complete the function compute_error(), shown in Figure 6, to compute the sum of squared errors (residuals) for scoring the model, using the equation we discussed in class. Here, we normalise this based on the number of instances in the training set. This is so that we can compare the training error with the evaluation error.

$$S(\omega) = \frac{1}{M}\sum_{j=1}^{M}(y_j - \hat{y}_j)^2 \tag{2}$$

```
#--
# gradient_descent_2()
# This function solves linear regression with gradient descent for 2
# parameters.
# inputs:
#  M = number of instances
#  $\chi$ = list of variable values for M instances
#  $\omega$ = list of parameters values (of size 2)
#  y = list of target values for M instances
#  $\alpha$ = learning rate
# output:
#  w = updated list of parameter values
#---
def gradient_descent_2( M, $\chi$, $\omega$, y, $\alpha$ ):
    for j ← 1, M
        $\hat{y} \leftarrow w_0 + w_1 * x_j$ # predict target value
        $\varepsilon \leftarrow y_j - \hat{y}$ # compare prediction to observation
        # adjust parameters:
        $w_0 \leftarrow w_0 + \alpha * \varepsilon * \frac{1}{M}$
        $w_1 \leftarrow w_1 + \alpha * \varepsilon * x_j * \frac{1}{M}$
    return $\omega$
```

Figure 5: Pseudocode for the gradient descent algorithm

```
#---
# compute_error()
# This function computes the sum of squared errors for the model.
# inputs:
#  M = number of instances
#  $\chi$ = list of variable values for M instances
#  $\omega$ = list of parameters values (of size 2)
#  y = list of target values
# output:
#  error (scalar)
#---
def compute_error( M, $\chi$, $\omega$, y ):
    error ← 0
    for j ← 1, M
        $\hat{y} \leftarrow w_0 + w_1 * x_j$ # predict target value
        error ← error + $(y_j - \hat{y})^2$ # compare prediction to observation
    error ← error / M
    return error
```

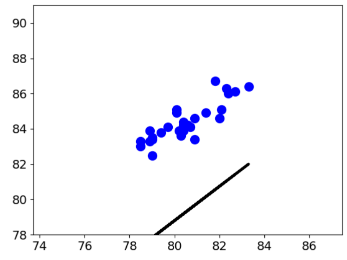
Figure 6: Pseudocode for computing the sum of squared errors

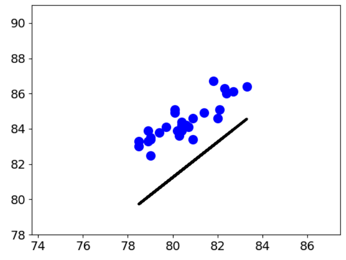
3. Now, to test your linear regression algorithm to see if one is a good predictor of the other. Use the **London Borough Profiles** data set (see part 3.1). Run your code with a learning rate of $\alpha = 0.001$ for 100 iterations and see if your regression equation will converge.

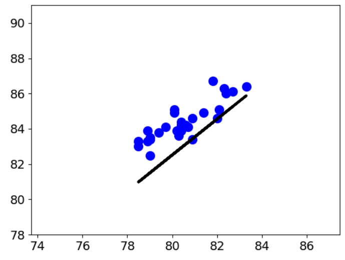
   I recommend plotting the regression line periodically, to see if your solution is converging, as shown in Figure 7.

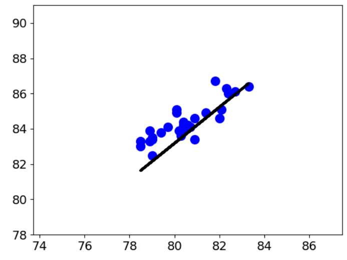
4. Next, let's compute a standard score for your model to evaluate how good it is overall—not just relative to previous sets of parameter values. The R^2 score is commonly used for evaluating the quality of a prediction. The equation is:

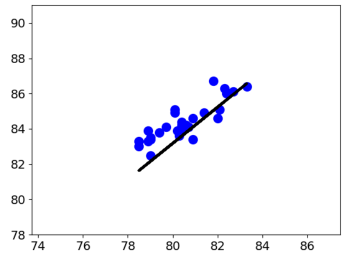
$$R^2 = 1 - (u/v)$$


(a) after 1 iteration
$error = 126.125$
$R^2 = -98.132$


(b) after 2 iterations
$error = 34.491$
$R^2 = -26.110$


(c) after 3 iterations
$error = 9.622$
$R^2 = -6.563$


(d) after 4 iterations
$error = 2.905$
$R^2 = -1.283$


(e) done training
$error = 0.485$
$R^2 = 0.619$

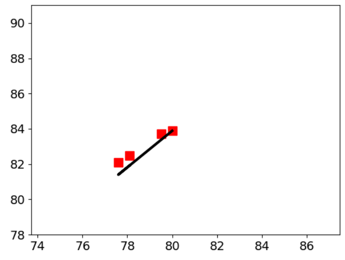

(f) testing results
$error = 0.693$
$R^2 = 0.268$

Figure 7: Progression of Linear Regression on London Borough Profiles data set. Sum of squared errors and R^2 scores are shown.

where

$$u = \sum_{j=1}^{M} (y_j - \hat{y_j})^2$$

which should look familiar as the sum of squared errors; and

$$v = \sum_{j=1}^{M} (y_j - y_{mean})^2$$

which is the population variance with a uniform distribution.

If the model is perfect, then u will be 0 and R^2 will be 1. If the model is terrible, then R^2 will be negative.

→ Write one more function to compute R^2, using the equation above. To help you out, pseudocode is shown in Figure 8.

5. How good is your model when scored using R^2? As your model iterates, your R^2 score should improve (approach 1.0), something like the plot shown in Figure 9.

6. Finally, use a **synthetic** data set (see part 3.3). Note that I had to iterate many more times with the synthetic data for the solution to converge (more than 10,000 iterations). My results are shown in Figure 10.

```
#---
# compute_r2()
# This function computes $R^2$ for the model.
# inputs:
#  M = number of instances
#  $\chi$ = list of variable values for M instances
#  $\omega$ = list of parameters values (of size 2)
#  y = list of target values
# output:
#  R^2 (scalar)
#---
def compute_r2( M, $\chi$, $\omega$, y ):
    $u = \sum_{j=1}^{M}(y_j - \hat{y_j})^2$
    $v = \sum_{j=1}^{M}(y_j - y_{mean})^2$
    $R^2 = 1 - (u/v)$
    return $R^2$
```

Figure 8: Pseudocode for computing R^2

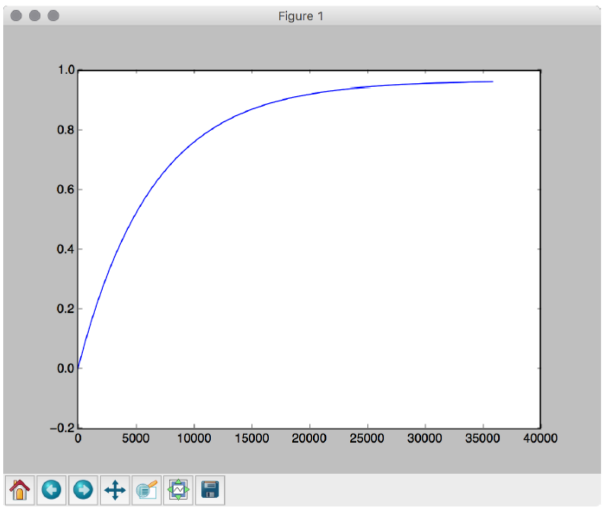
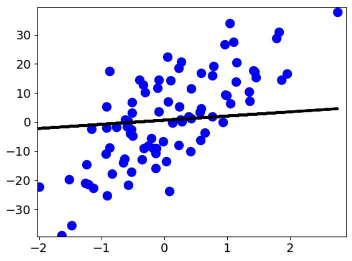
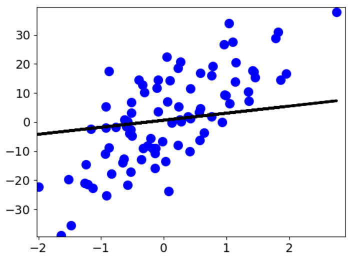
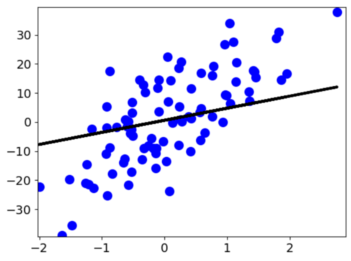


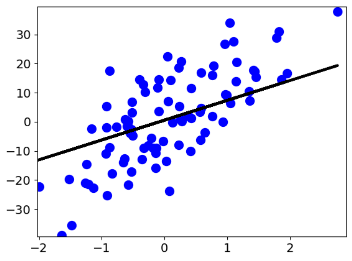
Figure 9: Example results for R^2 score

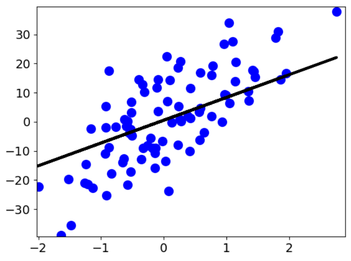
Try generating several different synthetic data sets, with different values for the noise. The higher the noise value, the more irregular the data is. How does raising the noise level effect the number of times it takes for your regression algorithm to converge?


(a) after 1000 iterations
$error = 5809.770$
$R^2 = 0.184$


(b) after 2000 iterations
$error = 4591.491$
$R^2 = 0.355$


(c) after 4000 iterations
$error = 2890.085$
$R^2 = 0.594$


(d) after 8000 iterations
$error = 1193.740$
$R^2 = 0.832$


(e) done training
$error = 789.892$
$R^2 = 0.889$

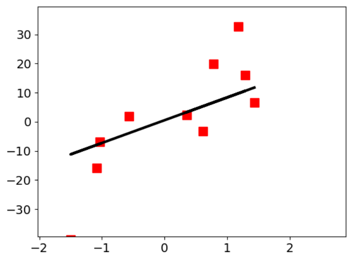

(f) testing results
$error = 990.282$
$R^2 = 0.879$

Figure 10: Progression of Linear Regression on synthetic data set. Sum of squared errors and R^2 scores are shown.

4.1.1 Choosing a Learning Rate

Note that selecting a good learning rate is the main challenge with the gradient descent algorithm. If the learning rate (α) is too small, then the gradient descent update rule will make very tiny updates on each step and it will take a lot of steps to converge. In contrast, if the learning rate is too big, then the gradient descent will make large jumps and risk "jumping over" the local minimum.

Both situations can be indicated by measuring the value of the objective function $S(\omega)$ on each step of gradient descent. In the first case (i.e., the learning rate is too small), the objective function will take a lot of steps (iterations) to converge to the minimum value. The convergence can be observed when the value of the objective function doesn't decrease significantly in consecutive steps. The second situation (i.e., learning rate is too large) can be indicated when the objective function suddenly increases after some step of the gradient descent.

Play around with different values of the learning rate—e.g., $\alpha = 0.0001, 0.01, 0.1, ...$ and see whether you can find a good value such that the minimum of the objective function is reached in the least number of steps without "jumping over" it.

Also, in your experiments, try some large values of *alpha*, just so you can see what the "jumping over" behaviour looks like.

For each selected value of learning rate, plot how the value of R^2 changes with each iteration. Once a good learning rate is found, report the final ω inferred with gradient descent and compare it with the "ground truth" value that was used to generate the data.

4.1.2 Knowing when to stop iterating

In your code, you iterated for a fixed number of times (i.e., 100). But, as we discussed in class, it is hard to know exactly how many times to iterate. A better stopping condition is detecting *convergence*. One indication of this is when the objective function $S(\omega)$ stops changing significantly from one iteration to the next.

Modify your code so that, instead of iterating for a fixed number of times, you iterate until the solution has converged—i.e., until the error rate from one iteration to the next changes by less than a small threshold amount (you can decide what that value is). Report how many times you iterated. Is it more or less than 100 ?

Conduct additional experiments using different values of α and see how that changes the number of iterations you need for convergence.

4.2 Using the Linear Regression package in scikit-learn

Now that you have written your own code to perform gradient descent, you should understand better how it works. So we can move on to using the **LinearRegression** package that comes with **scikit-learn**. You can read about it here:

`http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html`

1. Write a new version of your code that uses the scikit-learn LinearRegression() function. The syntax for calling the package is listed in Figure 11.

```
from sklearn import linear_model
from sklearn import metrics
#-initialise model
lr = linear_model.LinearRegression()
#-build model by fitting parameters to the training data
lr.fit( x_train, y_train )
#-output the regression equation
print 'scikit regression equation: y = ',
print lr.intercept_,
print ' + ',
print lr.coef_[0],
print 'x'
#-measure how good the predictions are with the training data and output scores
y_hat = lr.predict( x_train )
print 'r2 = ',
print metrics.r2_score( y_train, y_hat )
print 'mean squared error = ',
print metrics.mean_squared_error( y_train, y_hat )
#-measure how good the predictions are with the test data and output scores
y_hat = lr.predict( x_test )
print 'r2 = ',
print metrics.r2_score( y_test, y_hat )
print 'mean squared error = ',
print metrics.mean_squared_error( y_test, y_hat )
```

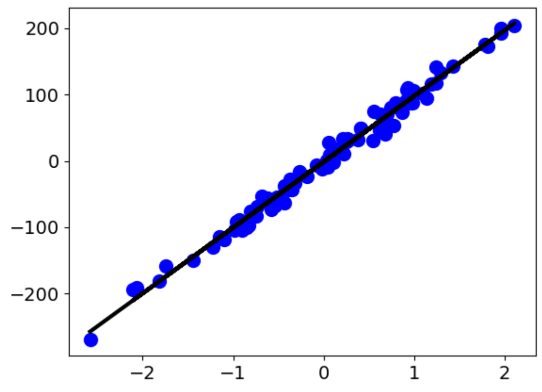
Figure 11: Example code for invoking LinearRegression function from scikit-learn

2. Compare your results to those obtained when using the scikit-learn package. Try it with both the downloaded and the synthetic data. Some of my results are shown in Figure 12.

   Note: Use the **sklearn.metrics** package for computing scores, as it seems that the score () function in the LinearRegression package does not return the correct value. The **metrics** package is here:

   `http://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html#sklearn.metrics.r2_score`

   Use the r2_score () and mean_squared_error() functions to report regression metrics.


(a) done training
$error = 98.977$
$R^2 = 0.985$

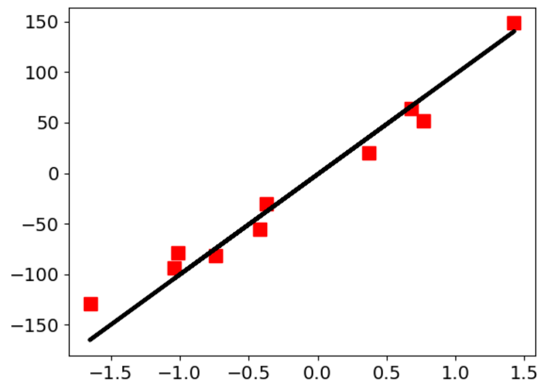

(b) testing results
$error = 105.981$
$R^2 = 0.984$

Figure 12: Results of **scikit-learn** Linear Regression on synthetic data set. Mean squared error and R^2 scores are shown.

5 Linear Classification with a Perceptron

This last exercise introduces the *Perceptron* class in **scikit-learn**:

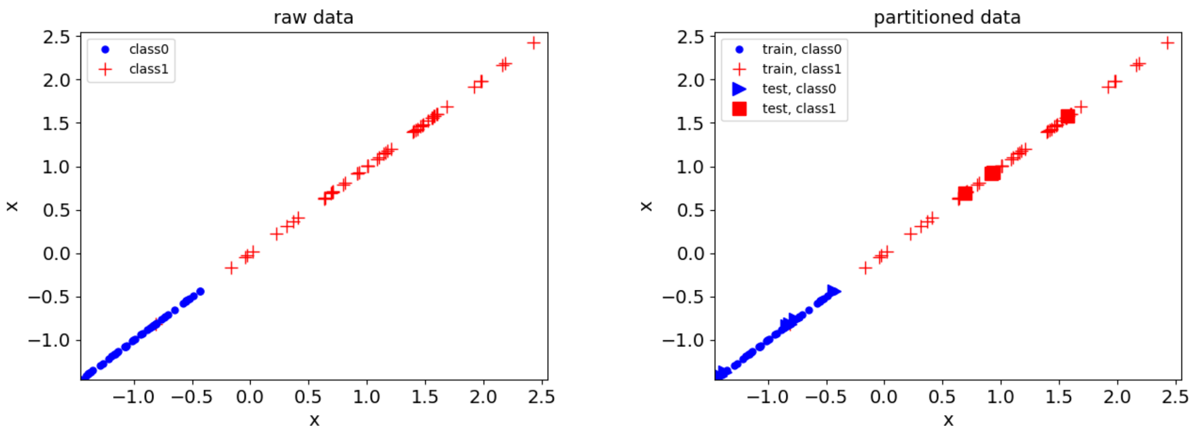
http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html

1. First, you need to generate a new synthetic data set designed for classification. Example code is shown in Figure 13. Note that this data set only has one **feature** (or variable or attribute), so it will be easier to visualise. There is one **label** (or class) which can take on one of two values—this is specified by the `n_classes=2` parameter in the function call.

```python
#--
# generate some random data for classification
#--
from sklearn import datasets
x, y = datasets.make_classification( n_features=1, n_redundant=0, n_informative
    =1, n_classes=2, n_clusters_per_class=1, n_samples=100 )
```

Figure 13: Example code for generating synthetic data for classification

2. As you did earlier, plot the raw data. Since you only have one feature, plot each x_j against itself. Since you have two classes, you can colour the markers for each class using a different colour and shape. As shown in Figure 2a, I used blue circles for class 0 and red Xs for class 1.



(a) raw data (b) partitioned data

Figure 14: Example synthetic data set for classification

3. Next, initialise a Perceptron model and fit it to your data. Example code is shown in Figure 15, including code to plot the decision boundary after the perceptron is built.

   My example output is shown in Figure 16. Note that plotting like this only works nicely as a 2D plot because I have used only one feature. If we used two features, we could plot those against each other (e.g., x_0 vs x_1 and colour the markers according to class), but there would then be a plane separating the classes in feature space and that is harder to visualise.

```
from sklearn import linear_model
per = linear_model.Perceptron() # initialise the perceptron model
per.fit( x, y ) # build the model by adjusting the weights to fit the data
y_hat = per.predict( x ) # predict some outputs using the model
y_plot[j] = per.intercept_ + x[j] * per.coef_[0,0] ) # plot the decision boundary
```

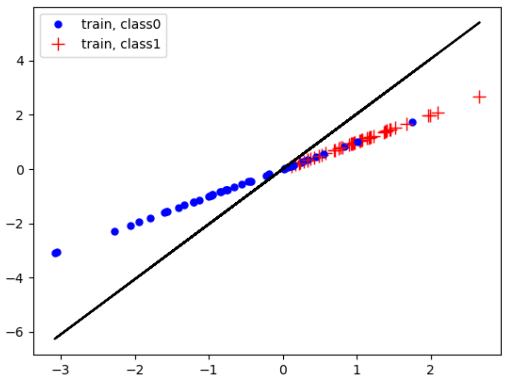
Figure 15: Example code for Linear Classification using a Perceptron

4. Finally, report your results. Use the **sklearn.metrics** package which has a number of classification metrics:

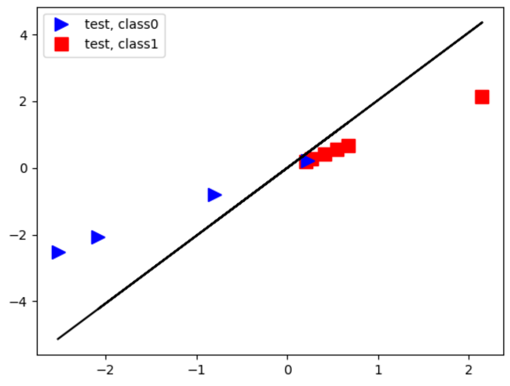
   `http://scikit-learn.org/stable/modules/model_evaluation.html#classification-metrics`

   Try using the `metrics.accuracy_score()` function, which computes the percentage of correctly predicted labels:

```
from sklearn import metrics
print 'accuracy = %f' % ( metrics.accuracy_score( y, y_hat, normalize=True )
    )
```



(a) done training
$accuracy = 0.967$



(b) testing results
$accuracy = 0.900$

Figure 16: Example output showing decision boundary computed by perceptron model