To begin with, let us load the scikit learn library and import the module that contains the various functions in order to extract the inbuilt datasets

In [**5**]: from sklearn.datasets import load\_iris,load\_boston,make\_classification, make\_circles, make\_moons

First, we will look into the ‘Iris flower data set’ by Ronald Fisher.

This dataset would be used for classification problem.

The load\_iris function returns a dictionary object

In [**6**]: data = load\_iris()

The predictor x, response variable y, response variable names and feature names can be extracted by querying the dictionary object with the appropriate keys

In [**7**]: x = data['data']

In [**8**]: y = data['target']

In [**9**]: y\_labels = data['target\_names']

In [**10**]: x\_labels = data['feature\_names']

Let’s print to see the values

In [**11**]: print

In [**12**]: print x.shape

(150, 4)

In [**13**]: print y.shape

(150,)

In [**14**]: print x\_labels

['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

In [**15**]: print y\_labels

['setosa' 'versicolor' 'virginica']

We can see that predictors have 150 instances and 4 attributes.

The response variable has 150 instances and a class label for each of the rows in our predictor set.

We are then printing out the attribute names, petal and sepal width and length, and finally, the class labels.

Now we will see some features with the Boston Housing Dataset.

This dataset would be used for regression problem.

The data is loaded pretty much the same way as is done above for iris flower data set.

In [**16**]: data = load\_boston()

The various components of the data, including the predictors and response variables, are queried using the respective keys from the dictionary.

In [**17**]: x = data['data']

In [**18**]: y = data['target']

In [**19**]: x\_labels = data['feature\_names']

Let's print these variables to understand more.

In [**20**]: print

In [**21**]: print x.shape

(506, 13)

In [**22**]: print y.shape

(506,)

In [**23**]: print x\_labels

['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'

'B' 'LSTAT']

We can see that the predictor set x has 506 instances and 13 attributes.

The response variable has 506 entries.

Finally, we have also printed out the names of the attributes.

Scikit-learn has also provided functions that will help us produce a random classification dataset with some desired properties

# make some classification dataset

The make\_classification function is a function that can be used to generate a classification dataset.

In [**24**]: x,y = make\_classification(n\_samples = 50, n\_features = 5, n\_classes = 2)

In this example, we generated a dataset with 50 instances that are dictated by the n\_samples parameter, five attributes, n\_features parameters, and two classes set by the n\_classes parameter.

Let’s print to analyze the output of this function.

In [**25**]: print

In [**26**]: print x.shape

(50, 5)

In [**27**]: print y.shape

(50,)

In [**28**]: print x[1,:]

[-0.06371735 0.24424535 -0.3383024 0.84448991 0.94607483]

In [**29**]: print y[1]

0

We can see that predictor x has 150 instances with 5 features.

The response variable has 150 instances, with a class label for each of the prediction instances.

On printing the second record in our predictor set x, we can see that we have a vector of dimension 5, relating to the five features that we requested.

Also, we will print the response variable y. For the second row of our predictors, the class label is 0.

Scikit learn also provide us with functions that can generate data with non-linear relationships.

In [**30**]: x,y = make\_circles()

In [**31**]: import numpy as np

In [**32**]: import matplotlib.pyplot as plt

In [**33**]: plt.close('all')

In [**34**]: plt.figure(1)

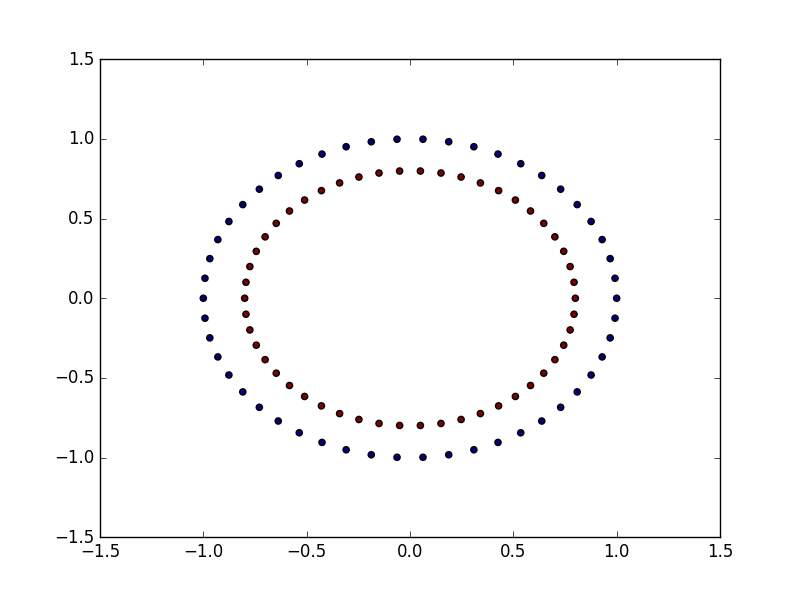
Out[**34**]: <matplotlib.figure.Figure at 0x9a589b0>

In [**35**]: plt.scatter(x[:,0],x[:,1],c=y)

Out[**35**]: <matplotlib.collections.PathCollection at 0x9b8b3f0>

In [**36**]: plt.show()

Let us look at the plot to understand non – linear relationship.



Our classification has produced two concentric circles. x is a dataset with two variables. Variable y is the class label. As shown by the concentric circle, the relationship between our prediction variable is nonlinear.

Another interesting function to produce a nonlinear relationship is make\_moons from scikit-learn.

In [**39**]: x,y = make\_moons()

In [**40**]: plt.figure(2)

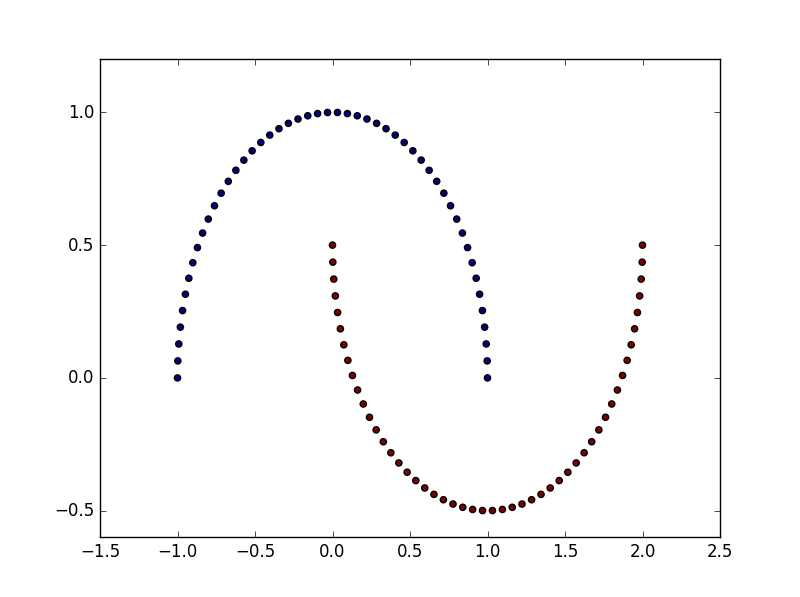
Out[**40**]: <matplotlib.figure.Figure at 0x9a8c150>

In [**42**]: plt.scatter(x[:,0],x[:,1],c=y)

Out[**42**]: <matplotlib.collections.PathCollection at 0x9d4cd10>

In [**43**]: plt.show()

Lets look at the plot to understand more.



The crescent-shaped plot shows that the attributes in our predictor set x are nonlinearly related to each other.

Let us now learn the API structures of Scikit learn. One of the major advantages of using scikit-learn is its clean API structure. All the data modeling classes deriving from the BaseEstimator class have to strictly implement the fit and transform functions. We will see some examples regarding this.

Lets see how we can put some machine learning functionalities in scikit learn

In [**44**]: import numpy as np

Lets start with the preprocessing module of scikit learn

In [**45**]: from sklearn.preprocessing import PolynomialFeatures

We will use the PolynomialFeatures class in order to demonstrate the ease of using scikit-learn's SDK.

With a set of predictor variables, we may want to add some more variables to our predictor set in order to see if our model accuracy can be improved. We can use the polynomials of the existing features as a new feature. The PolynomialFeatures class helps us do this.

We will first create a dataset. In this case, our dataset has two instances and two attributes –

In [**46**]: x = np.asmatrix([[1,2],[2,4]])

We will proceed to instantiate our PolynomialFeatures class with the required degree of polynomials. In this case, it will be a second degree

In [**47**]: poly = PolynomialFeatures(degree = 2)

There are two functions, fit and transform. The fit function is used to do the necessary calculations for the transformation. The transform function takes the input and, based on the calculations performed by fit, transforms the given input.

In [**48**]: poly.fit(x)

Out[**48**]: PolynomialFeatures(degree=2, include\_bias=True, interaction\_only=False)

In [**49**]: x\_poly = poly.transform(x)

In [**50**]: print "Original x variable shape", x.shape

Original x variable shape (2, 2)

#original x values

In [**51**]: print x

[[1 2]

[2 4]]

In [**52**]: print

In [**53**]: print "Transformed x variables", x\_poly.shape

Transformed x variables (2, 6)

# transformed x values

In [**54**]: print x\_poly

[[ 1. 1. 2. 1. 2. 4.]

[ 1. 2. 4. 4. 8. 16.]]

Alternatively, in this case, fit and transform can be called in one shot and the output would be the same.

In [**55**]: x\_poly = poly.fit\_transform(x)

Any class that implements a machine learning method in scikit-learn has to deliver from BaseEstimator.

BaseEstimator expects that the implementation class provides both the fit and transform methods. This way the API is kept very clean.

Lets see another example. Here we imported a class called DecisionTreeClassifier from the module tree. DecisionTreeClassifier implements the decision tree algorithm

In [**56**]: from sklearn.tree import DecisionTreeClassifier

In [**57**]: from sklearn.datasets import load\_iris

In [**58**]: data = load\_iris()

In [**59**]: x = data['data']

In [**60**]: y = data['target']

In [**61**]: estimator = DecisionTreeClassifier()

In [**62**]: estimator.fit(x,y)

Out[**62**]:

DecisionTreeClassifier(class\_weight=None, criterion='gini', max\_depth=None,

max\_features=None, max\_leaf\_nodes=None, min\_samples\_leaf=1,

min\_samples\_split=2, min\_weight\_fraction\_leaf=0.0,

presort=False, random\_state=None, splitter='best')

In [**63**]: predicted\_y = estimator.predict(x)

In [**64**]: predicted\_y\_prob = estimator.predict\_proba(x)

In [**65**]: predicted\_y\_lprob = estimator.predict\_log\_proba(x)

C:\Users\Sukanya\Anaconda2\lib\site-packages\sklearn\tree\tree.py:715: RuntimeWarning: divide by zero encountered in log

return np.log(proba)

We have used the iris dataset to see how the tree algorithm can be used. First, we have loaded the iris dataset in the x and y variables. We then instantiate DecisonTreeClassifier. We proceeded to build the model by invoking the fit function and passing our x predictor and y response variable. This built the tree model.

Now, we are ready with our model to do some predictions. We have used the predict function in order to predict the class labels for the given input. As we can see, we leveraged the same fit and predict method as in PolynomialFeatures. There are two other methods, predict\_proba - which gives the probability of the prediction, and predict\_log\_proba – which provides the logarithm of the prediction probability.

We will see another interesting utility called pipelining. Various machine learning methods can be chained together using pipe lining.

In [**66**]: from sklearn.pipeline import Pipeline

In [**67**]: poly = PolynomialFeatures(n=3)

In [**70**]: tree\_estimator = DecisionTreeClassifier()

Let's start by instantiating the data processing routines, PolynomialFeatures and DecisionTreeClassifier

In [**71**]: steps = [('poly',poly),('tree',tree\_estimator)]

We will define a list of tuples to indicate the order of our chaining. We want to run the polynomial feature generation, followed by our decision tree

In [**72**]: estimator = Pipeline(steps = steps)

In [**73**]: estimator.fit(x,y)

Out[**73**]:

Pipeline(steps=[('poly', PolynomialFeatures(degree=2, include\_bias=True, interaction\_only=False)), ('tree', DecisionTreeClassifier(class\_weight=None, criterion='gini', max\_depth=None,

max\_features=None, max\_leaf\_nodes=None, min\_samples\_leaf=1,

min\_samples\_split=2, min\_weight\_fraction\_leaf=0.0,

presort=False, random\_state=None, splitter='best'))])

In [**74**]: predicted\_y = estimator.predict(x)

We can now instantiate our Pipeline object with the list declared using the steps variable. Now, we can proceed as usual by calling the fit and predict methods.

We can invoke the named\_steps attribute in order to inspect the models in the various stages of our pipeline.

In [**75**]: estimator.named\_steps

Out[**75**]:

{'poly': PolynomialFeatures(degree=2, include\_bias=True, interaction\_only=False),

'tree': DecisionTreeClassifier(class\_weight=None, criterion='gini', max\_depth=None,

max\_features=None, max\_leaf\_nodes=None, min\_samples\_leaf=1,

min\_samples\_split=2, min\_weight\_fraction\_leaf=0.0,

presort=False, random\_state=None, splitter='best')}