HSMA_ML

September 22, 2018

1 Logistic regression

1.1 Load data

This example uses a data set built into sklearn. It classifies biopsy samples from breast cancer patients as ether malignant (cancer) or benign (no cancer).

In [2]:
```
from sklearn import datasets

        data_set = datasets.load_breast_cancer()

        # Set up features (X), labels (y) and feature names
        X = data_set.data
        y = data_set.target
        feature_names = data_set.feature_names
        label_names = data_set.target_names
```

Show feature names.

In [3]:
```
print (feature_names)
```

```
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

Show first record feature data.

In [4]:
```
print (X[1])
```

```
[2.057e+01 1.777e+01 1.329e+02 1.326e+03 8.474e-02 7.864e-02 8.690e-02
 7.017e-02 1.812e-01 5.667e-02 5.435e-01 7.339e-01 3.398e+00 7.408e+01
 5.225e-03 1.308e-02 1.860e-02 1.340e-02 1.389e-02 3.532e-03 2.499e+01
```

```
 2.341e+01 1.588e+02 1.956e+03 1.238e-01 1.866e-01 2.416e-01 1.860e-01
 2.750e-01 8.902e-02]
```

Show label names.

```
In [5]: print (label_names)
```

```
['malignant' 'benign']
```

Print first 25 lables.

```
In [6]: print (y[:25])
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0]
```

We are dealing with a binary outcome. There are just two possibilities: benign or malignant. The methods described below will also work with problems with more than two possible classifications, but we'll keep things relatively simple here.

1.2 Split the data into training and test sets

Data will be split randomly with 75% of the data used to train the machine learning model, and 25% used to test the model.

```
In [7]: from sklearn.model_selection import train_test_split

        X_train,X_test,y_train,y_test=train_test_split(
                X,y,test_size=0.25)
```

1.3 Scale the data using standardisation

We scale data so that all features share similar scales.

The X data will be transformed by standardisation. To standardise we subtract the mean and divide by the standard deviation. All data (training + test) will be standardised using the mean and standard deviation of the training set.

We will use a scaler from sklearn (but we could do this manually).

```
In [8]: from sklearn.preprocessing import StandardScaler

        # Initialise a new scaling object for normalising input data
        sc=StandardScaler()

        # Set up the scaler just on the training set
        sc.fit(X_train)

        # Apply the scaler to the training and test sets
        X_train_std=sc.transform(X_train)
        X_test_std=sc.transform(X_test)
```

Look at the first row of the raw and scaled data.

```
In [9]: print ('Raw data:')
        print (X_train[0])
        print ()
        print ('Scaled data')
        print (X_train_std[0])
```

```
Raw data:
[1.288e+01 1.822e+01 8.445e+01 4.931e+02 1.218e-01 1.661e-01 4.825e-02
 5.303e-02 1.709e-01 7.253e-02 4.426e-01 1.169e+00 3.176e+00 3.437e+01
 5.273e-03 2.329e-02 1.405e-02 1.244e-02 1.816e-02 3.299e-03 1.505e+01
 2.437e+01 9.931e+01 6.747e+02 1.456e-01 2.961e-01 1.246e-01 1.096e-01
 2.582e-01 8.893e-02]

Scaled data
[-0.36737431 -0.24219862 -0.32226491 -0.4700164   1.83425665  1.21709729
 -0.52651885  0.09871505 -0.35972527  1.39174201  0.15619514 -0.05861438
  0.17858324 -0.11883777 -0.57112622 -0.1052128  -0.61143715  0.10804629
 -0.24792232 -0.17943819 -0.26773858 -0.22328682 -0.25111534 -0.37512237
  0.55131326  0.25668614 -0.74440738 -0.10266883 -0.50914117  0.25431606]
```

1.4 Fit logistic regression model

Our first machien learning model is a logistic regression model.
https://en.wikipedia.org/wiki/Logistic_regression

```
In [10]: from sklearn.linear_model import LogisticRegression

         ml = LogisticRegression(C=1000)
         ml.fit(X_train_std,y_train)
```

```
Out[10]: LogisticRegression(C=1000, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)
```

That's it! We can now use the model to predict malignant vs. benign classification of patients.

```
In [11]: # Predict training and test set labels
         y_pred_train = ml.predict(X_train_std)
         y_pred_test = ml.predict(X_test_std)
```

1.5 Check accuracy of model

```
In [12]: import numpy as np

         accuracy_train = np.mean(y_pred_train == y_train)
```

```
accuracy_test = np.mean(y_pred_test == y_test)

print ('Accuracy of prediciting training data =', accuracy_train)
print ('Accuracy of prediciting test data =', accuracy_test)
```

```
Accuracy of prediciting training data = 0.9929577464788732
Accuracy of prediciting test data = 0.958041958041958
```

Notice that the accuracy of fitting training data is significantly higher than the test data. This is known as over-fitting. There are two potential problems with over-fitting:

1) If we test accuracy on the same data used to train the model we report a spuriously high accuracy. Our model is not actually as good as we report.
2) The model is too tightly built around our training data.

The solution to the first problem is to always report accuracy based on predicting the values of a test data set that was not used to train the model.

The solution to the second problem is to use 'regularisation'.

1.6 Regularisation

Regularisation 'loosens' the fit to the training data. It effectively moves all predictions a little closer to the average for all values.

In our logistic regression model, the regularisation parameter is C. A C value of 1,000 means there is very little regularisation. Try changing the values down by factors of 10, re-run code blocks 9, 10 and 11 and see what happens to the accuracy of the model. What value of C do you think is best?

1.7 Examinining the probability of outcome, and changing the sensitivity of the model to predicting a positive

There may be cases where either:

1) We want to see the probability of a given classification, or
2) We want to adjust the sensitivity of predicting a certain outcome (e.g. for health screening we may choose to accept more false positives in order to minimise the number fo false negatives).

For linear regression we use the output 'predict_proba'. This may also be used in other machine learning models such as random forests and neural networks (but for support vector machines the output 'decision_function' is used in place of predict_proba.

Let's look at it in action.

For this section we'll refit the model with regularisation of C=0.1.

```
In [13]: # We'll start by retraining the model with C=0.1

         ml = LogisticRegression(C=0.1)
         ml.fit(X_train_std,y_train)
```

```
        y_pred_test = ml.predict(X_test_std)

        # Calculate the predicted probability of outcome 1:

        y_pred_probability = ml.predict_proba(X_test_std)

        # Print first 5 values and the predicted label:

        print ('Predicted label probabilities:')
        print (y_pred_probability[0:5])
        print ()
        print ('Predicted labels:')
        print (y_pred_test[0:5])
        print ()
        print ('Actual labels:')
        print (y_test[0:5])
```

```
Predicted label probabilities:
[[9.99999977e-01 2.27009500e-08]
 [2.31989267e-01 7.68010733e-01]
 [5.99718849e-02 9.40028115e-01]
 [1.58608410e-02 9.84139159e-01]
 [1.87433182e-03 9.98125668e-01]]

Predicted labels:
[0 1 1 1 1]

Actual labels:
[0 1 1 1 1]
```

Let's calculate false positive and false negatives . In this data set being clear of chance has a label '1', and having cancer has a label '0'. A false positive, that is a sample is classed as cancer when is not actually cancer has a predicted test label of 0 and an actual label of 0. A false negative (predicted no cancer when cancer is actually present) has a predicted label of 1 and and actual label of zero.

```
In [14]: fp = np.sum((y_pred_test == 1) & (y_test == 0))
         fn = np.sum((y_pred_test == 0) & (y_test == 1))

         print ('False positives:', fp)
         print ('False negatives:', fn)
```

```
False positives: 2
False negatives: 0
```

Maybe we are more concerned about false negatives. Let's adjust the probability cut-off to change the threshold for classification as having no cancer (predicted label 1).

```
In [15]: cutoff = 0.75

         # Now let's make a prediction based on that new cutoff.
         # Column 1 contains the probability of no cancer

         new_prediction = y_pred_probability[:,1] >= cutoff
```

And let's calculate our false positives and negatives:

```
In [16]: fp = np.sum((new_prediction == 0) & (y_test == 1))
         fn = np.sum((new_prediction == 1) & (y_test == 0))

         print ('False positives:', fp)
         print ('False negatives:', fn)
```

```
False positives: 8
False negatives: 1
```

We have eliminated false negatives, but at the cost of more false postives. Try adjusting the cuttoff value. What value do you think is best?

1.8 Model weights (coefficients)

We can obtain the model weights (coefficients) for each of the features. Values that are more strongly positive or negative are most important. A positive number means that this feature is linked to a classification label of 1. A negative number means that this feature is linked to a classification label of 0. We can obtain the weights by using the method .coef_ (be careful to add the trailing underscore).

```
In [17]: print (ml.coef_)
```

```
[[-0.34349044 -0.33178015 -0.33844463 -0.36708645 -0.12187052  0.00765482
  -0.42255581 -0.40341256 -0.07540728  0.20606778 -0.44130835  0.01567654
  -0.33062184 -0.3657388  -0.0561784   0.19323076 -0.01636882 -0.01956193
   0.06890254  0.24517315 -0.53861132 -0.52077952 -0.50924417 -0.51384138
  -0.38831286 -0.11229711 -0.41778047 -0.43760388 -0.3899574  -0.09911312]]
```

2 Random Forests

A second type of categorisation model is a Random Forest.
https://en.wikipedia.org/wiki/Random_forests

```
In [18]: from sklearn.ensemble import RandomForestClassifier

         ml = RandomForestClassifier(n_estimators = 10000,
                                     n_jobs = -1)

         # For random forests we don't need to use scaled data
         ml.fit (X_train,y_train)
```

```
Out[18]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                     max_depth=None, max_features='auto', max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=10000, n_jobs=-1,
                     oob_score=False, random_state=None, verbose=0,
                     warm_start=False)
```

```
In [19]: # Predict test set labels

         y_pred_test = ml.predict(X_test)
         accuracy_test = np.mean(y_pred_test == y_test)
         print ('Accuracy of prediciting test data =', accuracy_test)
```

```
Accuracy of prediciting test data = 0.9440559440559441
```

2.1 Feature importance

Feature importances give us the relative importance of each feature - the higher the number the greater the influence on the decision (they add up to 1.0). Feature importances do not tell use which decision is more likely.

(Careful to add the trailing underscore in ml.feature_importances_)

```
In [20]: import pandas as pd

         df = pd.DataFrame()
         df['feature'] = feature_names
         df['importance'] = ml.feature_importances_
         df = df.sort_values('importance', ascending = False)
         print (df)
```

```
                   feature  importance
22          worst perimeter    0.145793
20             worst radius    0.126224
23               worst area    0.120934
27     worst concave points    0.107566
7       mean concave points    0.086282
6            mean concavity    0.050089
2            mean perimeter    0.049308
3                 mean area    0.046792
0               mean radius    0.041615
26          worst concavity    0.036971
13               area error    0.035772
25        worst compactness    0.016429
21            worst texture    0.015007
10             radius error    0.014587
12          perimeter error    0.012707
1              mean texture    0.012507
```

```
28           worst symmetry    0.011385
24         worst smoothness    0.010911
5          mean compactness    0.010349
29  worst fractal dimension    0.006442
4           mean smoothness    0.005170
11            texture error    0.005052
16          concavity error    0.004991
17     concave points error    0.004303
15        compactness error    0.004159
19  fractal dimension error    0.004147
18           symmetry error    0.003951
14         smoothness error    0.003708
8             mean symmetry    0.003454
9    mean fractal dimension    0.003395
```

3 ADDITIONAL MATERIAL

4 Support Vector Machines

https://en.wikipedia.org/wiki/Support_vector_machine

Support Vector Machines are another common classification algorithm. Regularisation (C) may be adjusted, and different 'kernels' may also be applied. The two most common are 'linear' and 'rbf').

In [21]:
```
# Import data

from sklearn import datasets

data_set = datasets.load_breast_cancer()

# Set up features (X), labels (y) and feature names

X = data_set.data
y = data_set.target
feature_names = data_set.feature_names
label_names = data_set.target_names

# Split data into training and test sets

from sklearn.model_selection import train_test_split

X_train,X_test,y_train,y_test=train_test_split(
        X,y,test_size=0.25)

# Scale data
```

```
from sklearn.preprocessing import StandardScaler
sc=StandardScaler()
sc.fit(X_train)
X_train_std=sc.transform(X_train)
X_test_std=sc.transform(X_test)

# Fit model
# Note: a common test is to see whether linear or rbf kernel is best
# Try changing regularisation (C)

from sklearn.svm import SVC
ml = SVC(kernel='linear',C=100)
ml.fit(X_train_std,y_train)

# Predict training and test set labels

y_pred_train = ml.predict(X_train_std)
y_pred_test = ml.predict(X_test_std)

# Check accuracy of model

import numpy as np
accuracy_train = np.mean(y_pred_train == y_train)
accuracy_test = np.mean(y_pred_test == y_test)
print ('Accuracy of prediciting training data =', accuracy_train)
print ('Accuracy of prediciting test data =', accuracy_test)

# Show classification probabilities for first five samples
# Note that for SVMs we use decision_function, in place of predict_proba

y_pred_probability = ml.decision_function(X_test_std)
print ()
print ('Predicted label probabilities:')
print (y_pred_probability[0:5])
```

```
Accuracy of prediciting training data = 0.9953051643192489
Accuracy of prediciting test data = 0.972027972027972

Predicted label probabilities:
[-0.7905539   6.09369495 16.11186543  6.55771422 21.89224697]
```

5 Neural Networks

Neural networks may be better for very large or complex data sets. A challenge is the number of parameters that need to be optimised.

After importing the MLPClassifier (another name for a Neural Network is a Mulit-Level Perceptron Classifier) type help (MLPClassifier) for more information.

In [22]:
```
# Import data

from sklearn import datasets

data_set = datasets.load_breast_cancer()

# Set up features (X), labels (y) and feature names

X = data_set.data
y = data_set.target
feature_names = data_set.feature_names
label_names = data_set.target_names

# Split data into training and test sets

from sklearn.model_selection import train_test_split

X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.25)

# Scale data

from sklearn.preprocessing import StandardScaler
sc=StandardScaler()
sc.fit(X_train)
X_train_std=sc.transform(X_train)
X_test_std=sc.transform(X_test)

# Fit model
# Note: a common test is to see whether linear or rbf kernel is best
# Try changing regularisation (C)

from sklearn.neural_network import MLPClassifier

ml = MLPClassifier(solver='lbfgs',
                   alpha=1e-8,
                   hidden_layer_sizes=(50, 10),
                   max_iter=100000,
                   shuffle=True,
                   learning_rate_init=0.001,
                   activation='relu',
                   learning_rate='constant',
                   tol=1e-7,
                   random_state=0)

ml.fit(X_train_std, y_train)

# Predict training and test set labels
```

```
y_pred_train = ml.predict(X_train_std)
y_pred_test = ml.predict(X_test_std)

# Check accuracy of model

import numpy as np
accuracy_train = np.mean(y_pred_train == y_train)
accuracy_test = np.mean(y_pred_test == y_test)
print ('Accuracy of prediciting training data =', accuracy_train)
print ('Accuracy of prediciting test data =', accuracy_test)

# Show classification probabilities for first five samples
# Neural networks may often produce spuriously high proabilities!

y_pred_probability = ml.predict_proba(X_test_std)
print ()
print ('Predicted label probabilities:')
print (y_pred_probability[0:5])
```

```
Accuracy of prediciting training data = 1.0
Accuracy of prediciting test data = 0.972027972027972

Predicted label probabilities:
[[0.00000000e+00 1.00000000e+00]
 [0.00000000e+00 1.00000000e+00]
 [1.00000000e+00 2.25849723e-60]
 [1.00000000e+00 1.40199390e-70]
 [3.77475828e-15 1.00000000e+00]]
```

6 A Random Forest example with multiple categories

We will use another classic 'toy' data set to look at multiple label classification. This is the categorisation of iris plants. We only have four features but have three different classification possibilities. We will use logistic regression, but all the methods described above work on multiple label classification. Note: for completeness of code we'll import the required modules again, but this is not actually necessary if they have been imported previously.

6.1 Load data

In [23]:
```
from sklearn import datasets

data_set = datasets.load_iris()

X = data_set.data
y = data_set.target
feature_names = data_set.feature_names
label_names = data_set.target_names
```

```
print ('Label names:')
print (label_names)
print ()
print ('Feature names:')
print (feature_names)
print ()
print ('First sample feature values:')
print (X[0])
print ()
print ('Labels:')
print (y)
```

```
Label names:
['setosa' 'versicolor' 'virginica']

Feature names:
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

First sample feature values:
[5.1 3.5 1.4 0.2]

Labels:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

6.2 Split data into training and test data sets

```
In [24]: from sklearn.model_selection import train_test_split

         X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.25)
```

6.3 Scale data

Here we'll use a random forest model which does not need scaling. But for all other model types a scaling step would be added here.

6.4 Fit random forest model and show accuracy

```
In [25]: ml = RandomForestClassifier(n_estimators = 10000,
                                     random_state = 0,
                                     n_jobs = -1)

         # For random forests we don't need to use scaled data
         ml.fit (X_train,y_train)
```

```
Out[25]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                     max_depth=None, max_features='auto', max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=10000, n_jobs=-1,
                     oob_score=False, random_state=0, verbose=0, warm_start=False)
```

6.5 Predict training and test set labels

```
In [26]: y_pred_train = ml.predict(X_train)
         y_pred_test = ml.predict(X_test)
```

6.6 Check accuracy of model

```
In [27]: import numpy as np

         accuracy_train = np.mean(y_pred_train == y_train)
         accuracy_test = np.mean(y_pred_test == y_test)

         print ('Accuracy of prediciting training data =', accuracy_train)
         print ('Accuracy of prediciting test data =', accuracy_test)
```

```
Accuracy of prediciting training data = 1.0
Accuracy of prediciting test data = 0.9736842105263158
```

6.7 Show classification of first 10 samples

```
In [28]: print ('Actual label:')
         print (y_test[0:10])
         print ()
         print ('Predicted label:')
         print (y_pred_test[0:10])
```

```
Actual label:
[2 0 2 1 2 2 0 0 1 1]

Predicted label:
[1 0 2 1 2 2 0 0 1 1]
```

6.8 Showing prediction probabilities and changing sensitivity to classification

As with a binary classification we may be interested in obtaining the probability of label classification, either to get an indicator of the certainty of classification, or to bias classification towards or against particular classes.

Changing sensitivity towards particular class labels is more complicated with multi-class problems, but the principle is the same as with a binary classification. We can access the calculated

probabilities of classification for each label. Usually the one with the highest probability is taken, but rules could be defined to bias decisions more towards one class if that is beneficial.

Here we will just look at the probability outcomes for each class. The usual rule for prediction is to simply take the one that is highest.

```
In [29]: y_pred_probability = ml.predict_proba(X_test)

         print (y_pred_probability[0:5])
```

```
[[0.000e+00 8.058e-01 1.942e-01]
 [9.997e-01 1.000e-04 2.000e-04]
 [0.000e+00 1.100e-03 9.989e-01]
 [1.000e-04 9.988e-01 1.100e-03]
 [0.000e+00 2.200e-03 9.978e-01]]
```