Doing Classification

Derek Bridge  
School of Computer Science and Information Technology  
University College Cork

Initialization

```
In [2]: %load_ext autoreload
   %autoreload 2
   %matplotlib inline
```

```
In [3]: import pandas as pd
   import numpy as np
   import matplotlib.pyplot as plt
```

Doing Classification

We'll revisit the topics of Data Preparation and Error Estimation to see whether any of the details change now that we're looking at classification instead of regression.

Data Preparation
You must consider all the things that we discussed before:

- **Getting to know your data**
- **Detecting anomalies**
- **Handling missing values**

We discussed some of the approaches: revisit the data collection process; delete examples; delete features; and impute values.

When we imputed values, we used the mean (numeric-valued features) or the mode (nominal-valued features). If example \( \langle x, y \rangle \) has no value for feature \( x_j \), we impute the mean (or mode) from (training) examples \( x' \) where \( x'_j \) does have a value. But, in classification, there is a variation on these.

We could instead use the class mean, or class mode. In other words, if example \( \langle x, y \rangle \) has no value for feature \( x_j \), we impute the mean (or mode) but only from (training) examples \( \langle x', y \rangle \) where \( x'_j \) has a value.

- **Handling nominal-valued features**

Remember that some classifiers require that the dependent variable \( y \) be numeric, e.g. 1 = ham and 0 = spam. If in the original dataset it is not numeric, then you will need to encode it. Do not use one-hot encoding for this; instead, simply convert to numbers.

So in scikit-learn, for \( y \) you would use LabelEncoder, but not OneHotEncoder.

**Question:**
- In the previous module, we discussed an exception to this. What was it?

- **Scaling numeric values**

One really good way of getting to know your data is with a visualization.

### Simple Visualizations

Given a dataset, we often want to see how values of \( y \) are distributed with respect to values of one of the features. We can use the following:

- Scatter plots;
- Histograms; or
- Box plots.

### Scatter Plots

For regression, we have used **scatter plots** with \( y \) on the vertical axis and the feature on the horizontal axis, e.g.:
In [4]: df = pd.read_csv("dataset-corkA.csv")

    x = df[['flarea']].values
    y = df[['price']].values

    fig = plt.figure()
    plt.title("Price against floor area")
    plt.xlabel("Floor area (sq metres)")
    plt.xlim(0, 500)
    plt.ylabel("Price (000 euros)"
    plt.ylim(0, 1000)
    plt.scatter(x, y, color = 'green')
    plt.show()

What about datasets for classification? A scatter plot can be useful but they look a bit weird because only two values on the vertical axis are possible:
```
In [5]: df = pd.read_csv("dataset-cs1109.csv")
    
x = df[['lect']].values
y = df[['outcome']].values

fig = plt.figure()
plt.title("Outcome against lecture attendance")
plt.xlabel("Lecture attendance (%)")
plt.xlim(0, 100)
plt.ylabel("Outcome (0 = fail, 1 = pass)"")
plt.ylim(-0.1, 1.1)
plt.scatter(x, y, color = 'green')
plt.show()
```

Better might be histograms.

**Histograms**
Another option is **box plots**.

**Box Plots**
A box plot shows:

- The bottom of the box is the **first quartile**
  - 25% of the data has values lower than this
- The red line inside the box is the second quartile (the **median**)
  - 50% of the data has values lower than this
- The top of the box is the **third quartile**
  - 75% of the data has lower values than this
- The difference between the third quartile and the first quartile is called the **inter-quartile range** (IQR)
- By default, the **whiskers** show the lowest value still within 1.5 IQR of the first quartile, and the highest value still within 1.5 IQR of the third quartile
- Any other values ('outliers' or 'fliers') beyond the whiskers are shown individually

## Error Estimation

### Accuracy
The first question is what to measure.

For regression, we measured mean squared error (MSE). Alternatives include root mean-squared error, mean absolute error, relative squared error, root relative squared error, relative absolute error, and others. Most of the time, if you’re comparing one regressor with another, it doesn’t much matter which of these you use: often, the same ‘winner’ emerges no matter which metric you use.

For classification, there is a lot of choice, which we will discuss at greater length in a separate lecture. For the purposes of this lecture, we will use error rate, which is simply the proportion of the test set for which we made the correct classification. If you like notation:

\[
\frac{1}{|T|} \sum_{i=1}^{|T|} I(\hat{y}^{(i)} \neq y^{(i)})
\]

where \(T\) is the set of examples on which you are testing the classifier and hence \(|T|\) is the number of examples, and \(I(p)\) is the indicator function that outputs 1 if predicate \(p\) is true and zero otherwise.

More often, we report the classification accuracy. It’s simply the proportion of correct classifications made over a set of \(m\) examples:

\[
\frac{1}{|T|} \sum_{i=1}^{|T|} I(\hat{y}^{(i)} = y^{(i)})
\]

Classification accuracy is simply the complement of error rate (1 minus the error rate).

### Training Sets and Test Sets

With the exception of leave-one-out-cross-validation (LOOCV), error estimation relies on randomly partitioning the dataset into a training set and a test set. We’ve discussed before that the split may be ‘lucky’ or ‘unlucky’. One example of this, in the case of classification, is that the split might not reflect the distribution of examples within the classes. For example, examples of one class might be under-represented in the training set or test set — examples of one class might even be completely absent from the training set or test set.

**Stratification** is the solution to this: in stratification, the proportion of examples of each class in the overall dataset is respected in the partitioning into training and test sets. Here’s pseudocode for stratified holdout. For simplicity, the pseudocode only covers the case of binary classification:

- Divide the dataset into positive examples, \(P\), and negative examples, \(N\)
- Randomly partition \(P\) into \(Train_P\) and \(Test_P\)
- Randomly partition \(N\) into \(Train_N\) and \(Test_N\)
- \(Train \leftarrow Train_P \cup Train_N\)
- \(Test \leftarrow Test_P \cup Test_N\)

Although this fixes the distribution with respect to the classes, you may still get ‘lucky’ or ‘unlucky’ in other ways. So you will still want to do the above multiple times — stratified repeated holdout.

In scikit-learn, use StratifiedShuffleSplit, which is similar to ShuffleSplit, except that it takes a different first argument: instead of the size of the dataset, it takes in the array of \(y\)-values (so that it can respect their distribution):
In [8]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.cross_validation import StratifiedShuffleSplit
from sklearn.cross_validation import cross_val_score

# Read CSV file
df = pd.read_csv("dataset-cs1109.csv")

# Get the feature-values into a separate numpy arrays of numbers and
# the target values into a separate
# numpy arrays of ints
X = df[['lec', 'lab', 'cao']].values
y = df['outcome'].values

sss = StratifiedShuffleSplit(y, n_iter = 10, test_size = 0.3, random
_state = np.random)
estimator = LogisticRegression()
accuracies_test = cross_val_score(estimator, X, y, scoring = 'accuracy', cv = sss)
mean_accuracy_test = np.mean(accuracies_test)
mean_accuracy_test
```

Out[8]: 0.7621359223309697

Note the use of 'accuracy' as the scoring function in the above.

Often better still than stratified repeated holdout is stratified \textit{k-fold cross-validation}. In the case of binary classification, the dataset is again divided into $P$ and $N$. You then partition $P$ into $k$ folds, and similarly $N$. You then pair up folds from $P$ with folds from $N$, resulting in $k$ stratified folds.

Here it is in scikit-learn:

In [9]:
```
from sklearn.cross_validation import StratifiedKFold
from sklearn.cross_validation import cross_val_score

skf = StratifiedKFold(y, n_folds = 10) # No shuffle or random_state
_parameters!
estimator = LogisticRegression()
accuracies_test = cross_val_score(estimator, X, y, scoring = 'accuracy', cv = skf)
mean_accuracy_test = np.mean(accuracies_test)
mean_accuracy_test
```

Out[9]: 0.75159663865546222

Remember the shorthand way of doing \textit{k-fold cross validation}, simply by setting \texttt{cv} to an integer? The same
works here: when you're doing classification, it will give you stratified \textit{k-fold cross-validation}:

In [10]:
```
accuracies_test = cross_val_score(estimator, X, y, scoring = 'accuracy', cv = 10)
mean_accuracy_test = np.mean(accuracies_test)
mean_accuracy_test
```

Out[10]: 0.75159663865546222
Note that an accuracy of 76% accuracy is OK but not great on this dataset. A majority-class classifier (one that always predicts pass) would get 60%.

## Model Selection

Both the classifiers that we have looked at so far have hyperparameters. For kNN, it is $k$; for logistic regression, it is $\lambda$, which controls the regularization. Optimal values are found, again, with, e.g. grid search.

When it came to regularized linear regression (Lasso and Ridge), scikit-learn made it easy to do model selection using the LassoCV and RidgeCV classes. For logistic regression, it seems we have to explicitly use GridSearch. Furthermore, the parameter that controls the regularization (called C) is the inverse of $\lambda$ — small values result in increased regularization.

```python
In [11]: from sklearn.cross_validation import KFold
from sklearn.grid_search import GridSearchCV

estimator_l1 = LogisticRegression(penalty = 'l1')
estimator_l2 = LogisticRegression(penalty = 'l2') # default
parameters = {'C': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}

gs_l1 = GridSearchCV(estimator_l1, parameters, scoring = 'accuracy',
cv = 10)
accuracies_test_l1 = cross_val_score(gs_l1, X, y, scoring = 'accuracy',
cv = 10)
mean_accuracies_test_l1 = np.mean(accuracies_test_l1)
gs_l2 = GridSearchCV(estimator_l2, parameters, scoring = 'accuracy',
cv = 10)
accuracies_test_l2 = cross_val_score(gs_l2, X, y, scoring = 'accuracy',
cv = 10)
mean_accuracies_test_l2 = np.mean(accuracies_test_l2)
print(mean_accuracies_test_l1, mean_accuracies_test_l2)

(0.78949579831932781, 0.78957983193277292)
```

Note that this is using stratified $k$-fold cross-validation — because we used $cv = 10$. 