05a_accuracy_logistic_regression

December 23, 2019

1 Kaggle Titanic survival - logistic regression model

In this notebook we repeat our basic logistic regression model as previously described:

https://github.com/MichaelAllen1966/1804_python_healthcare/blob/master/titanic/02_logistic_regression.ipyn

We will extend the model to report a range of accuracy measures, as described:

https://github.com/MichaelAllen1966/1804_python_healthcare/blob/master/titanic/05_accuracy_standalone.ip

We will go through the following steps:

- Download and save pre-processed data
- Split data into features (X) and label (y)
- Split data into training and test sets (we will test on data that has not been used to fit the model)
- Standardise data
- Fit a logistic regression model (from sklearn learn)
- Predict survival of the test set
- Define a function to calculate a range of accuracy measure (and return as a dictionary)
- Report multipmultipleel accuracy scores for model

1.1 Load modules

A standard Anaconda install of Python (https://www.anaconda.com/distribution/) contains all the necessary modules.

[1]:
```
import numpy as np
import pandas as pd
# Import machine learning methods
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

1.2 Load data

The section below downloads pre-processed data, and saves it to a subfolder (from where this code is run). If data has already been downloaded that cell may be skipped.

Code that was used to pre-process the data ready for machine learning may be found at: https://github.com/MichaelAllen1966/1804_python_healthcare/blob/master/titanic/01_preprocessing.ipynb

[2]:
```
download_required = True

if download_required:

    # Download processed data:
    address = 'https://raw.githubusercontent.com/MichaelAllen1966/' + \
                '1804_python_healthcare/master/titanic/data/processed_data.csv'

    data = pd.read_csv(address)

    # Create a data subfolder if one does not already exist
    import os
    data_directory ='./data/'
    if not os.path.exists(data_directory):
        os.makedirs(data_directory)

    # Save data
    data.to_csv(data_directory + 'processed_data.csv')
```

[3]:
```
data = pd.read_csv('data/processed_data.csv')
```

The first column is a passenger index number. We will remove this, as this is not part of the original Titanic passenger data.

[4]:
```
# Drop Passengerid (axis=1 indicates we are removing a column rather than a row)
# We drop passenger ID as it is not original data

data.drop('PassengerId', inplace=True, axis=1)
```

1.3 Divide into X (features) and y (lables)

We will separate out our features (the data we use to make a prediction) from our label (what we are truing to predict). By convention our features are called `X` (usually upper case to denote multiple features), and the label (survvive or not) `y`.

[5]:
```
X = data.drop('Survived',axis=1) # X = all 'data' except the 'survived' column
y = data['Survived'] # y = 'survived' column from 'data'
```

1.4 Divide into training and tets sets

When we test a machine learning model we should always test it on data that has not been used to train the model. We will use sklearn's `train_test_split` method to randomly split the data: 75% for training, and 25% for testing.

[6]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
```

1.5 Standardise data

We want all of out features to be on roughly the same scale. This generally leads to a better model, and also allows us to more easily compare the importance of different features.

One simple method is to scale all features 0-1 (by subtracting the minimum value for each value, and dividing by the new remaining maximum value).

But a more common method used in many machine learning methods is standardisation, where we use the mean and standard deviation of the training set of data to normalise the data. We subtract the mean of the test set values, and divide by the standard deviation of the training data. Note that the mean and standard deviation of the training data are used to standardise the test set data as well.

Here we will use sklearn's `StandardScaler method`. This method also copes with problems we might otherwise have (such as if one feature has zero standard deviation in the training set).

[7]:
```
def standardise_data(X_train, X_test):

    # 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
    train_std=sc.transform(X_train)
    test_std=sc.transform(X_test)

    return train_std, test_std
```

[8]:
```
X_train_std, X_test_std = standardise_data(X_train, X_test)
```

1.6 Fit logistic regression model

Now we will fir a logistic regression model, using sklearns `LogisticRegression` method. Our machine learning model fitting is only two lines of code! By using the name `model` for ur logistic regression model we will make our model more interchangeable later on.

[9]:
```
model = LogisticRegression(solver='lbfgs')
model.fit(X_train_std,y_train)
```

[9]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
```

```
                    random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                    warm_start=False)
```

1.7 Predict values

Now we can use the trained model to predict survival. We will test the accuracy of both the training and test data sets.

[10]:
```
# Predict training and test set labels
y_pred_train = model.predict(X_train_std)
y_pred_test = model.predict(X_test_std)
```

1.8 Calculate accuracy

Here we define a function that will calculate a range of accuracy scores.

[11]:
```
def calculate_accuracy(observed, predicted):

    """
    Calculates a range of acuuracy scores from observed and predicted classes.

    Takes two list or NumPy arrays (observed class values, and predicted class
    values), and returns a dictionary of results.

     1) observed positive rate: proportion of observed cases that are +ve
     2) Predicted positive rate: proportion of predicted cases that are +ve
     3) observed negative rate: proportion of observed cases that are -ve
     4) Predicted neagtive rate: proportion of predicted cases that are -ve
     5) accuracy: proportion of predicted results that are correct
     6) precision: proportion of predicted +ve that are correct
     7) recall: proportion of true +ve correctly identified
     8) f1: harmonic mean of precision and recall
     9) sensitivity: Same as recall
    10) specificity: Proportion of true -ve identified:
    11) positive likelihood: increased probability of true +ve if test +ve
    12) negative likelihood: reduced probability of true +ve if test -ve
    13) false positive rate: proportion of false +ves in true -ve patients
    14) false negative rate: proportion of false -ves in true +ve patients
    15) true postive rate: Same as recall
    16) true negative rate
    17) positive predictive value: chance of true +ve if test +ve
    18) negative predictive value: chance of true -ve if test -ve

    """

    # Converts list to NumPy arrays
```

```
    if type(observed) == list:
        observed = np.array(observed)
    if type(predicted) == list:
        predicted = np.array(predicted)

    # Calculate accuracy scores
    observed_positives = observed == 1
    observed_negatives = observed == 0
    predicted_positives = predicted == 1
    predicted_negatives = predicted == 0

    true_positives = (predicted_positives == 1) & (observed_positives == 1)

    false_positives = (predicted_positives == 1) & (observed_positives == 0)

    true_negatives = (predicted_negatives == 1) & (observed_negatives == 1)

    accuracy = np.mean(predicted == observed)

    precision = (np.sum(true_positives) /
                 (np.sum(true_positives) + np.sum(false_positives)))

    recall = np.sum(true_positives) / np.sum(observed_positives)

    sensitivity = recall

    f1 = 2 * ((precision * recall) / (precision + recall))

    specificity = np.sum(true_negatives) / np.sum(observed_negatives)

    positive_likelihood = sensitivity / (1 - specificity)

    negative_likelihood = (1 - sensitivity) / specificity

    false_postive_rate = 1 - specificity

    false_negative_rate = 1 - sensitivity

    true_postive_rate = sensitivity

    true_negative_rate = specificity

    positive_predictive_value = (np.sum(true_positives) /
                                 np.sum(observed_positives))

    negative_predicitive_value = (np.sum(true_negatives) /
                                  np.sum(observed_positives))
```

```
    # Create dictionary for results, and add results
    results = dict()

    results['observed_positive_rate'] = np.mean(observed_positives)
    results['observed_negative_rate'] = np.mean(observed_negatives)
    results['predicted_positive_rate'] = np.mean(predicted_positives)
    results['predicted_negative_rate'] = np.mean(predicted_negatives)
    results['accuracy'] = accuracy
    results['precision'] = precision
    results['recall'] = recall
    results['f1'] = f1
    results['sensivity'] = sensitivity
    results['specificity'] = specificity
    results['positive_likelihood'] = positive_likelihood
    results['negative_likelihood'] = negative_likelihood
    results['false_postive_rate'] = false_postive_rate
    results['false_negative_rate'] = false_negative_rate
    results['true_postive_rate'] = true_postive_rate
    results['true_negative_rate'] = true_negative_rate
    results['positive_predictive_value'] = positive_predictive_value
    results['negative_predicitive_value'] = negative_predicitive_value

    return results
```

[12]:
```
# Call calculate_accuracy function
accuracy = calculate_accuracy(y_test, y_pred_test)

# Print results up to three decimal places
for key, value in accuracy.items():
    print (key, "{0:0.3}".format(value))
```

```
observed_positive_rate 0.381
observed_negative_rate 0.619
predicted_positive_rate 0.323
predicted_negative_rate 0.677
accuracy 0.78
precision 0.75
recall 0.635
f1 0.688
sensivity 0.635
specificity 0.87
positive_likelihood 4.87
negative_likelihood 0.419
false_postive_rate 0.13
false_negative_rate 0.365
true_postive_rate 0.635
```

```
true_negative_rate 0.87
positive_predictive_value 0.635
negative_predicitive_value 1.41
```

We can see from the accuracy scores that overall accuracy is about 80%, but that accuracy is imbalanced between survivors and no-survivers. We can see the model is biased towards predicting fewer survivors than actually occurred, and this gives higher specificity (the proportion of non-survivors correctly identified) than sensitivity (the proportion of survivors correctly identified). In the next notebook we will look at adjusting the balance between sensitivity and specificity.

Note: To keep this example simpel we have used a single random split between training and test data. A more thorough analysis would use repeated measurement using stratified k-fold validation (see https://github.com/MichaelAllen1966/1804_python_healthcare/blob/master/titanic/03_k_fold.ipynb).