17_random_forest

December 31, 2019

1 Kaggle Titanic survival - a Random Forest model

Our previous work has all been based on logistic regression, which is the most common 'standard model' against which all other models are compared.

In this notebook we swap out the logistic regression model for a Random Forest model. Random Forests are often chosen for classification based on structured data (i.e. when we have specific features of data, rather than unstructured data like a picture or a sound file).

Random Forests are based on constructing multiple decision trees, each of which sees only part of the data for each case, and only has limited 'branches'. Random Forests tend to be less prone to over-fitting than decision trees. For more on the basis of Random Forests see:

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

Note in this example how similar the code is to our previous logistic regression model. A couple of notable changes are:

- Data for Random Forest models do not need standardisation; we use the raw data.
- Rather than having coefficients, we output model 'importances' which reflect how influential a feature is in deciding classification. This is accessed through examining `model.feature_importances_`.

Here we will again use stratified K-fold validation to test the model performance. We will use default settings for the Random Forest model.

1.1 Load modules

[1]:
```python
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold
```

1.2 Download data

Run the following code if data for Titanic survival has not been previously downloaded.

[2]:
```python
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', index=False)
```

1.3 Load data

[3]:
```
# Load data & drop passenger ID
data = pd.read_csv('data/processed_data.csv')
data.drop('PassengerId', inplace=True, axis=1)

# Split data into two DataFrames
X_df = data.drop('Survived',axis=1)
y_df = data['Survived']

# Convert DataFrames to NumPy arrays
X = X_df.values
y = y_df.values
```

1.4 Define function to calulate accuracy

[4]:
```
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
```

1.5 Run the model with k-fold validation

[8]:
```
# Set up lists to hold results for each k-fold run
replicate_accuracy = []
replicate_precision = []
replicate_recall = []
replicate_f1 = []
```

```
replicate_predicted_positive_rate = []
replicate_observed_positive_rate = []

# Set up DataFrame for feature importances
importances = pd.DataFrame(index = list(X_df))

# Convert DataFrames to NumPy arrays
X = X_df.values
y = y_df.values

# Set up splits
number_of_splits = 10
skf = StratifiedKFold(n_splits = number_of_splits)
skf.get_n_splits(X, y)

# Loop through the k-fold splits
k_fold_count = 0

for train_index, test_index in skf.split(X, y):

    # Get X and Y train/test
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Set up and fit model (n_jobs=-1 uses all cores on a computer)
    model = RandomForestClassifier(n_jobs=-1)

    model.fit(X_train,y_train)

    # Predict test set labels and get accuracy scores
    y_pred_test = model.predict(X_test)
    accuracy_scores = calculate_accuracy(y_test, y_pred_test)
    replicate_accuracy.append(accuracy_scores['accuracy'])
    replicate_precision.append(accuracy_scores['precision'])
    replicate_recall.append(accuracy_scores['recall'])
    replicate_f1.append(accuracy_scores['f1'])
    replicate_predicted_positive_rate.append(
        accuracy_scores['predicted_positive_rate'])
    replicate_observed_positive_rate.append(
        accuracy_scores['observed_positive_rate'])

    # Record feature importances
    col_title = 'split_' + str(k_fold_count)
    importances[col_title] = model.feature_importances_
    k_fold_count +=1

# Transfer results to list and add to data frame
```

```
results = pd.Series()
results['accuracy'] = np.mean(replicate_accuracy)
results['precision'] = np.mean(replicate_precision)
results['recall'] = np.mean(replicate_recall)
results['f1'] = np.mean(replicate_f1)
results['predicted positive rate'] = np.mean(replicate_predicted_positive_rate)
results['observed positive rate'] = np.mean(replicate_observed_positive_rate)

# Get average of feature importances, and sort
importance_mean = importances.mean(axis=1)
importance_mean.sort_values(ascending=False, inplace=True)
```

1.6 Show results and feature importance

[9]:
```
results
```

[9]:
```
accuracy                   0.801423
precision                  0.747805
recall                     0.730924
f1                         0.736352
predicted positive rate    0.375943
observed positive rate     0.383833
dtype: float64
```

[10]:
```
importance_mean
```

[10]:
```
male                   0.232531
Fare                   0.218718
Age                    0.215120
Pclass                 0.058164
CabinNumber            0.056248
SibSp                  0.047701
Parch                  0.039698
CabinNumberImputed     0.018376
CabinLetterImputed     0.016551
AgeImputed             0.016459
CabinLetter_missing    0.016114
Embarked_S             0.015642
Embarked_C             0.013035
Embarked_Q             0.007959
CabinLetter_E          0.005723
CabinLetter_C          0.005667
CabinLetter_B          0.004911
CabinLetter_D          0.004296
CabinLetter_A          0.003754
CabinLetter_F          0.001752
CabinLetter_G          0.001063
```

```
CabinLetter_T          0.000219
EmbarkedImputed        0.000163
Embarked_missing       0.000136
dtype: float64
```

1.7 Observations

- Without any optimisation we observe accuracy similar to, or a little higher than, logistic regression.
- Unlike logistic regression we see a good balance between precision and recall, rather than a biad towards the majority class.
- Performance of the model may be tested and optimised as we previously did for logistic regression (e.g. construct ROC curves, perform feature selection, consider over-sampling or under-sampling as required, examine learning curves).

For furtehr notes on the sklearn Random Forest model (and which paramters may be fine-tuned, e.g. with random search or grid search as we did with the logistic regression model) please see the help pages for the Random Forest model, or refer to:

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html