1 Random forests

Random forest is a versatile machine learning method based on decision trees. One useful feature of random forests is that it is easy to obtain the relative importance of features. This may be used to help better understand what drives classification, and may also be used to reduce the feature set used with minimal reduction in accuracy.

Once again we will re-use our logistic regression model, and replace the model function wit the following three lines:

from sklearn.ensemble import RandomForestClassifier model = RandomForestClassifier(n_estimators=10000, random_state=0, n_jobs=-1) model.fit (X,y)

We will also add a function to print out relative importance of features. These are obtained using ¡em¿model.feature_importances_¡/em¿.

Unlike many other methods, random forests do not need data to be normalised (though it won't cause any problems if you do).

```
# import required modules

from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np
import pandas as pd

def calculate_diagnostic_performance (actual_predicted):
    """ Calculate diagnostic performance.

    Takes a Numpy array of 1 and zero, two columns: actual and predicted

    Note that some statistics are repeats with different names
    (precision = positive_predictive_value and recall = sensitivity).
    Both names are returned

    Returns a dictionary of results:

    1) accuracy: proportion of test results that are correct
    2) sensitivity: proportion of true +ve identified
    3) specificity: proportion of true -ve identified
    4) positive likelihood: increased probability of true +ve if test +ve
    5) negative likelihood: reduced probability of true +ve if test -ve
    6) false positive rate: proportion of false +ves in true -ve patients
    7) false negative rate:  proportion of false -ves in true +ve patients
    8) positive predictive value: chance of true +ve if test +ve
    9) negative predictive value: chance of true -ve if test -ve
    10) precision = positive predictive value
    11) recall = sensitivity
    12) f1 = (2 * precision * recall) / (precision + recall)
    13) positive rate = rate of true +ve (not strictly a performance measure)
    """
    # Calculate results
    actual_positives = actual_predicted[:, 0] == 1
    actual_negatives = actual_predicted[:, 0] == 0
    test_positives = actual_predicted[:, 1] == 1
    test_negatives = actual_predicted[:, 1] == 0
    test_correct = actual_predicted[:, 0] == actual_predicted[:, 1]
    accuracy = np.average(test_correct)
    true_positives = actual_positives & test_positives
```

```
    true_negatives = actual_negatives & test_negatives
    sensitivity = np.sum(true_positives) / np.sum(actual_positives)
    specificity = np.sum(true_negatives) / np.sum(actual_negatives)
    positive_likelihood = sensitivity / (1 - specificity)
    negative_likelihood = (1 - sensitivity) / specificity
    false_positive_rate = 1 - specificity
    false_negative_rate = 1 - sensitivity
    positive_predictive_value = np.sum(true_positives) / np.sum(test_positives)
    negative_predictive_value = np.sum(true_negatives) / np.sum(test_negatives)
    precision = positive_predictive_value
    recall = sensitivity
    f1 = (2 * precision * recall) / (precision + recall)
    positive_rate = np.mean(actual_predicted[:,1])

    # Add results to dictionary
    performance = {}
    performance['accuracy'] = accuracy
    performance['sensitivity'] = sensitivity
    performance['specificity'] = specificity
    performance['positive_likelihood'] = positive_likelihood
    performance['negative_likelihood'] = negative_likelihood
    performance['false_positive_rate'] = false_positive_rate
    performance['false_negative_rate'] = false_negative_rate
    performance['positive_predictive_value'] = positive_predictive_value
    performance['negative_predictive_value'] = negative_predictive_value
    performance['precision'] = precision
    performance['recall'] = recall
    performance['f1'] = f1
    performance['positive_rate'] = positive_rate

    return performance

def load_data ():
    """Load the data set. Here we load the Breast Cancer Wisconsin (Diagnostic)
    Data Set. Data could be loaded from other sources though the structure
    should be compatible with thi sdata set, that is an object with the
    following attribtes:
        .data (holds feature data)
        .feature_names (holds feature titles)
        .target_names (holds outcome classification names)
        .target (holds classification as zero-based number)
        .DESCR (holds text-based description of data set)"""

    data_set = datasets.load_breast_cancer()
    return data_set

def normalise (X_train,X_test):
    """Normalise X data, so that training set has mean of zero and standard
    deviation of one"""

    # 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)
    return X_train_std, X_test_std
```

```
def print_diagnostic_results (performance):
    """Iterate through, and print, the performance metrics dictionary"""

    print('\nMachine learning diagnostic performance measures:')
    print('-------------------------------------------------')
    for key, value in performance.items():
        print (key,'= %0.3f' %value) # print 3 decimal places
    return

def print_feaure_importances (model, features):
    print ()
    print ('Feature importances:')
    print ('--------------------')
    df = pd.DataFrame()
    df['feature'] = features
    df['importance'] = model.feature_importances_
    df = df.sort_values('importance', ascending = False)
    print (df)
    return

def split_data (data_set, split=0.25):
    """Extract X and y data from data_set object, and split into tarining and
    test data. Split defaults to 75% training, 25% test if not other value
    passed to function"""

    X=data_set.data
    y=data_set.target
    X_train,X_test,y_train,y_test=train_test_split(
        X,y,test_size=split, random_state=0)
    return X_train,X_test,y_train,y_test

def test_model(model, X, y):
    """Return predicted y given X (attributes)"""

    y_pred = model.predict(X)
    test_results = np.vstack((y, y_pred)).T
    return test_results

def train_model (X, y):
    """Train the model """
    from sklearn.ensemble import RandomForestClassifier
    model = RandomForestClassifier(n_estimators=10000,
                                random_state=0,
                                n_jobs=-1)
    model.fit (X,y)
    return model

###### Main code #######

# Load data
data_set = load_data()

# Split data into training and test sets
X_train,X_test,y_train,y_test = split_data(data_set, 0.25)

# Normalise data (not needed for Random Forests)
```

```
# X_train_std, X_test_std = normalise(X_train,X_test)

# Train model
model = train_model(X_train, y_train)

# Produce results for test set
test_results = test_model(model, X_test, y_test)

# Measure performance of test set predictions
performance = calculate_diagnostic_performance(test_results)

# Print performance metrics
print_diagnostic_results(performance)

# Print feature importances
print_feaure_importances (model, data_set.feature_names)

OUT:

Machine learning diagnostic performance measures:
-------------------------------------------------
accuracy = 0.972
sensitivity = 0.967
specificity = 0.981
positive_likelihood = 51.233
negative_likelihood = 0.034
false_positive_rate = 0.019
false_negative_rate = 0.033
positive_predictive_value = 0.989
negative_predictive_value = 0.945
precision = 0.989
recall = 0.967
f1 = 0.978
positive_rate = 0.615

Feature importances:
--------------------
                    feature  importance
22          worst perimeter    0.128502
7       mean concave points    0.122821
27     worst concave points    0.121434
23               worst area    0.102040
20             worst radius    0.099788
6            mean concavity    0.054834
2            mean perimeter    0.046540
3                 mean area    0.039068
13               area error    0.039064
0               mean radius    0.036952
26          worst concavity    0.034057
25        worst compactness    0.017705
10             radius error    0.016905
21            worst texture    0.016596
1              mean texture    0.014762
5          mean compactness    0.013044
12          perimeter error    0.012706
24         worst smoothness    0.012666
28           worst symmetry    0.012059
29  worst fractal dimension    0.006879
```

16	concavity error	0.006393
4	mean smoothness	0.005968
17	concave points error	0.005948
19	fractal dimension error	0.005639
11	texture error	0.005319
15	compactness error	0.005317
8	mean symmetry	0.004822
18	symmetry error	0.004448
14	smoothness error	0.004105
9	mean fractal dimension	0.003617