0075_k_fold

April 20, 2018

1 Choosing between models with stratified k-fold validation

In previous example we have used multiple random sampling in order to obtain a better measurement of accuracy for modes (repeating the model with different random training/test splits).

A more robust method is to use 'stratified k-fold validation'. In this method the model is repeated k times, so that all the data is used once, but only once, as part of the test set. This, alone, is k-fold validation. Stratified k-fold validation adds an extra level of robustness by ensuring that in each of the k training/test splits, the balance of outcomes represents the balance of outcomes in the overall data set. Most commonly 10 different splits of the data are used.

In this example we shall load up some data on treatment of acute stroke (data will be loaded from the internet). The model will try to predict whether patients are treated with a clot-busting drug. We will compare a number of different models using straified k-fold validation.

```
In [35]: """Techniques applied:
             1. Random Forests
             2. Support Vector Machine (linear and rbf kernel)
             3. Logistic Regression
             4. Neural Network
         """

         # %% Load modules

         import os
         import numpy as np
         import pandas as pd
         from sklearn.model_selection import train_test_split
         from sklearn.preprocessing import StandardScaler
         from sklearn.ensemble import RandomForestClassifier
         from sklearn.svm import SVC
         from sklearn.linear_model import LogisticRegression
         from sklearn.neural_network import MLPClassifier
         from sklearn import datasets
         from sklearn.model_selection import StratifiedKFold


         # %% Function to calculate sensitivity ans specificty
         def calculate_diagnostic_performance(actual_predicted):
```

```python
    """ Calculate sensitivty and specificty.
    Takes a Numpy array of 1 and zero, two columns: actual and predicted
    Returns a tuple 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) Count of test positives

    *false positive rate is the percentage of healthy individuals who
    incorrectly receive a positive test result
    * alse neagtive rate is the percentage of diseased individuals who
    incorrectly receive a negative test result

    """
    actual_predicted = test_results.values
    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_postive_rate = 1 - specificity
    false_negative_rate = 1 - sensitivity
    positive_predictive_value = np.sum(true_positives) / np.sum(test_positives)
    negative_predicitive_value = np.sum(true_negatives) / np.sum(test_negatives)
    positive_rate = np.mean(actual_predicted[:,1])
    return (accuracy, sensitivity, specificity, positive_likelihood,
            negative_likelihood, false_postive_rate, false_negative_rate,
            positive_predictive_value, negative_predicitive_value,
            positive_rate)


# %% Print diagnostics results
def print_diagnostic_results(results):
    # format all results to three decimal places
    three_decimals = ["%.3f" % v for v in results]
    print()
    print('Diagnostic results')
    print('  accuracy:\t\t\t', three_decimals[0])
    print('  sensitivity:\t\t\t', three_decimals[1])
    print('  specificity:\t\t\t', three_decimals[2])
    print('  positive likelyhood:\t\t', three_decimals[3])
    print('  negative likelyhood:\t\t', three_decimals[4])
    print('  false positive rate:\t\t', three_decimals[5])
    print('  false negative rate:\t\t', three_decimals[6])
    print('  positive predictive value:\t', three_decimals[7])
    print('  negative predicitve value:\t', three_decimals[8])
    print()


# %% Calculate weights from weights ratio:
# Set up class weighting to bias for sensiitivty vs. specificity
# Higher values increase sensitivity at the cost of specificity
def calculate_class_weights(positive_class_weight_ratio):
    positive_weight = ( positive_class_weight_ratio /
                       (1 + positive_class_weight_ratio))

    negative_weight = 1 - positive_weight
    class_weights = {0: negative_weight, 1: positive_weight}
    return (class_weights)

#%% Create results folder if needed
# (Not used in this demo)
# OUTPUT_LOCATION = 'results'
# if not os.path.exists(OUTPUT_LOCATION):
#    os.makedirs(OUTPUT_LOCATION)

# %% Import data
url = ("https://raw.githubusercontent.com/MichaelAllen1966/wordpress_blog" +
       "/master/jupyter_notebooks/stroke.csv")
df_stroke = pd.read_csv(url)
feat_labels = list(df_stroke)[1:]
number_of_features = len(feat_labels)
X, y = df_stroke.iloc[:, 1:].values, df_stroke.iloc[:, 0].values

# Set different weights for pisitive and negative results in SVM is required
# This will adjust balance between sensitivity and specificity
# For equal weighting, set at 1
positive_class_weight_ratio = 1
class_weights = calculate_class_weights(positive_class_weight_ratio)

# Set up strtified k-fold
splits = 10
skf = StratifiedKFold(n_splits = splits)
```

```
skf.get_n_splits(X, y)

# %% Set up results dataframes
forest_results = np.zeros((splits, 10))
forest_importance = np.zeros((splits, number_of_features))
svm_results_linear = np.zeros((splits, 10))
svm_results_rbf = np.zeros((splits, 10))
lr_results = np.zeros((splits, 10))
nn_results = np.zeros((splits, 10))

# %% Loop through the k splits of training/test data
loop_count = 0

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

    print ('Split', loop_count + 1, 'out of', splits)

    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    sc = StandardScaler()  # new Standard Scalar object
    sc.fit(X_train)
    X_train_std = sc.transform(X_train)
    X_test_std = sc.transform(X_test)
    combined_results = pd.DataFrame()

    # %% Random forests
    forest = RandomForestClassifier(n_estimators=1000, n_jobs=-1,
                                    class_weight='balanced')
    forest.fit(X_train, y_train)
    forest_importance[loop_count, :] = forest.feature_importances_
    y_pred = forest.predict(X_test)
    test_results = pd.DataFrame(np.vstack((y_test, y_pred)).T)
    diagnostic_performance = (calculate_diagnostic_performance
                              (test_results.values))
    forest_results[loop_count, :] = diagnostic_performance
    combined_results['Forest'] = y_pred

    # %% SVM (Support Vector Machine) Linear
    svm = SVC(kernel='linear', C=1.0, class_weight=class_weights)
    svm.fit(X_train_std, y_train)
    y_pred = svm.predict(X_test_std)
    test_results = pd.DataFrame(np.vstack((y_test, y_pred)).T)
    diagnostic_performance = (calculate_diagnostic_performance
                              (test_results.values))
    svm_results_linear[loop_count, :] = diagnostic_performance
    combined_results['SVM_linear'] = y_pred
    # %% SVM (Support Vector Machine) RBF
    svm = SVC(kernel='rbf', C=1.0)
    svm.fit(X_train_std, y_train)
    y_pred = svm.predict(X_test_std)
    test_results = pd.DataFrame(np.vstack((y_test, y_pred)).T)
    diagnostic_performance = (calculate_diagnostic_performance
                              (test_results.values))
    svm_results_rbf[loop_count, :] = diagnostic_performance
    combined_results['SVM_rbf'] = y_pred

    # %% Logistic Regression
    lr = LogisticRegression(C=100, class_weight=class_weights)
    lr.fit(X_train_std, y_train)
    y_pred = lr.predict(X_test_std)
    test_results = pd.DataFrame(np.vstack((y_test, y_pred)).T)
    diagnostic_performance = (calculate_diagnostic_performance
                              (test_results.values))
    lr_results[loop_count, :] = diagnostic_performance
    combined_results['LR'] = y_pred

    # %% Neural Network
    clf = MLPClassifier(solver='lbfgs', alpha=1e-8, hidden_layer_sizes=(50, 5),
                        max_iter=100000, shuffle=True, learning_rate_init=0.001,
                        activation='relu', learning_rate='constant', tol=1e-7)
    clf.fit(X_train_std, y_train)
    y_pred = clf.predict(X_test_std)
    test_results = pd.DataFrame(np.vstack((y_test, y_pred)).T)
    diagnostic_performance = (calculate_diagnostic_performance
                              (test_results.values))
    nn_results[loop_count, :] = diagnostic_performance
    combined_results['NN'] = y_pred

    # Increment loop count
    loop_count += 1

# %% Transfer results to Pandas arrays
results_summary = pd.DataFrame()

results_column_names = (['accuracy', 'sensitivity',
                         'specificity',
                         'positive likelihood',
                         'negative likelihood',
                         'false positive rate',
                         'false negative rate',
                         'positive predictive value',
                         'negative predictive value',
                         'positive rate'])
```

```
        forest_results_df = pd.DataFrame(forest_results)
        forest_results_df.columns = results_column_names
        forest_importance_df = pd.DataFrame(forest_importance)
        forest_importance_df.columns = feat_labels
        results_summary['Forest'] = forest_results_df.mean()

        svm_results_lin_df = pd.DataFrame(svm_results_linear)
        svm_results_lin_df.columns = results_column_names
        results_summary['SVM_lin'] = svm_results_lin_df.mean()

        svm_results_rbf_df = pd.DataFrame(svm_results_rbf)
        svm_results_rbf_df.columns = results_column_names
        results_summary['SVM_rbf'] = svm_results_rbf_df.mean()

        lr_results_df = pd.DataFrame(lr_results)
        lr_results_df.columns = results_column_names
        results_summary['LR'] = lr_results_df.mean()

        nn_results_df = pd.DataFrame(nn_results)
        nn_results_df.columns = results_column_names
        results_summary['Neural'] = nn_results_df.mean()


        # %% Print summary results
        print()
        print('Results Summary:')
        print(results_summary)

        # %% Save files
        # NOT USED IN THIS DEMO
        # forest_results_df.to_csv('results/forest_results.csv')
        # forest_importance_df.to_csv('results/forest_importance.csv')
        # svm_results_lin_df.to_csv('results/svm_lin_results.csv')
        # svm_results_rbf_df.to_csv('results/svm_rbf_results.csv')
        # lr_results_df.to_csv('results/logistic_results.csv')
        # nn_results_df.to_csv('results/neural_network_results.csv')
        # results_summary.to_csv('results/results_summary.csv')
```

```
Split 1 out of 10
Split 2 out of 10
Split 3 out of 10
Split 4 out of 10
Split 5 out of 10
Split 6 out of 10
Split 7 out of 10
Split 8 out of 10
Split 9 out of 10
Split 10 out of 10
```

```
Results Summary:
                            Forest   SVM_lin   SVM_rbf        LR    Neural
accuracy                  0.851946  0.839995  0.843081  0.839610  0.801859
sensitivity               0.727978  0.767511  0.741951  0.753473  0.702353
specificity               0.905567  0.871350  0.886804  0.876865  0.844867
positive likelihood       8.799893  7.396559  7.384775  7.390298  4.909178
negative likelihood       0.297522  0.263269  0.287613  0.276478  0.349459
false positive rate       0.094433  0.128650  0.113196  0.123135  0.155133
false negative rate       0.272022  0.232489  0.258049  0.246527  0.297647
positive predictive value 0.775919  0.731363  0.747641  0.737093  0.669270
negative predictive value 0.887152  0.898619  0.890479  0.894471  0.869677
positive rate             0.285678  0.321505  0.302999  0.313414  0.320310
```