0119_grid_and_randomized_search

February 3, 2019

1 Optimising scikit-learn machine learning models with grid search or randomized search

Machine learning models have many hyper-parameters (parameters set before a model is fitted, and which remain constant throughout model fitting). Optimising model hyper-parameters may involve many model runs with alternative hyper-parameters. In SciKit-Learn, this may be performed in an automated fashion using Grid Search (which explores all combinations of provided hyper-parameters) or Randomized Search (which randomly selects combinations to test).

Grid search and randomized search will perform this optimisation using k-fold validation which avoids potential bias in training/test splits.

Here we will revisit a previous example of machine learning, using Random Forests to predict whether a person has breast cancer. We will then use Grid Search to optimise performance, using the 'f1' performance score (https://en.wikipedia.org/wiki/F1_score) as an accuracy score that balances the importance of false negatives and false positives.

First we will look at how we previously built the Random Forests model.

```
In [1]: # 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. Note n_jobs=-1 uses all cores on a computer"""
    from sklearn.ensemble import RandomForestClassifier
    model = RandomForestClassifier(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)
```

```
/home/michael/anaconda3/lib/python3.7/site-packages/sklearn/ensemble/forest.py:246: FutureWarn
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)



Machine learning diagnostic performance measures:
-------------------------------------------------
accuracy = 0.951
sensitivity = 0.956
specificity = 0.943
positive_likelihood = 16.881
negative_likelihood = 0.047
false_positive_rate = 0.057
false_negative_rate = 0.044
positive_predictive_value = 0.966
negative_predictive_value = 0.926
precision = 0.966
recall = 0.956
f1 = 0.961
positive_rate = 0.622
```

2 Optimise with grid search

NOTE: Grid search may take considerable time to run!

Grid search enables us to perform an exhaustive search of hyper-parameters (those model parameters that are constant in any one model). We define which hyper-parameters we wish to change, and what values we wish to try. All combinations are tested. Test are performed using k-fold validation which re-runs the model with different train/test splits (this avoids bias in our train/test split, but does increase the time required). You may wish to time small grid search first, so you have a better idea of how many parameter combinations you can realistically look at.

We pass four arguments to the grid search method:

1) The range of values for the hyper-parameters, defined in a dictionary
2) The machine learning model to use
3) The number of k-fold splits to use (`cv`); a value of 5 will give five 80:20 training/test splits with each sample being present in the test set once
4) The accuracy score to use. In a classification model `accuracy` is common. For a regression model using `scoring='neg_mean_squared_error'` is common (for grid search an accuracy score must be a 'utility function' rather than a 'cost function', that is, higher values are better).

If the best model uses a value at one extreme of the provided hyper-paramter ranges then it is best to expand the range of that hyper-paraemter to be sure an optimum has been found.

More info on grid search: https://scikit-learn.org/stable/modules/grid_search.html

An alternative approach is randomised hyper-parameter searching. See https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

```
In [2]: ## Use Grid search to optimise
        # n_jobs is set to -1 to use all cores on the CPU

        from sklearn.model_selection import GridSearchCV

        param_grid = {'n_estimators': [10, 30, 100, 300, 1000, 3000],
                      'bootstrap': [True, False],
                      'min_samples_split': [2, 4, 6, 8, 10],
                      'n_jobs': [-1]}

        # Grid search will use k-fold cross-validation (CV is number of splits)
        # Grid search also needs a ultility function (higher is better) rather than
        # a cost function (lower is better) so use neg square mean error

        from sklearn.ensemble import RandomForestClassifier
        forest_grid = RandomForestClassifier()
        grid_search = GridSearchCV(forest_grid, param_grid, cv=10,
                                   scoring='accuracy')

        grid_search.fit(X_train, y_train); #';' suppresses printed output
```

```
/home/michael/anaconda3/lib/python3.7/site-packages/sklearn/model_selection/_search.py:841: Dep
  DeprecationWarning)
```

Show optimised model hyper-parameters:

```
In [3]: # show best parameters
        # If best parameters are at the extremes of the searches then extend the range

        grid_search.best_params_

Out[3]: {'bootstrap': False, 'min_samples_split': 4, 'n_estimators': 30, 'n_jobs': -1}

In [4]: # Or, show full description
        grid_search.best_estimator_

Out[4]: RandomForestClassifier(bootstrap=False, class_weight=None, criterion='gini',
                    max_depth=None, max_features='auto', max_leaf_nodes=None,
                    min_impurity_decrease=0.0, min_impurity_split=None,
                    min_samples_leaf=1, min_samples_split=4,
                    min_weight_fraction_leaf=0.0, n_estimators=30, n_jobs=-1,
                    oob_score=False, random_state=None, verbose=0,
                    warm_start=False)
```

Now we will use the optimised model. We could use the text above (from the output of `grid_search.best_estimator_`, or we can use `grid_search.best_estimator_` directly.

```
In [5]: # Use optimised model
        model = grid_search.best_estimator_
        model.fit (X_train, y_train);
```

Test optimised model:

```
In [6]: 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)
```

```
Machine learning diagnostic performance measures:
-------------------------------------------------
accuracy = 0.972
sensitivity = 0.978
specificity = 0.962
positive_likelihood = 25.911
negative_likelihood = 0.023
false_positive_rate = 0.038
false_negative_rate = 0.022
positive_predictive_value = 0.978
negative_predictive_value = 0.962
precision = 0.978
recall = 0.978
f1 = 0.978
positive_rate = 0.629
```

Our accuracy has now increased from 95.1% to 97.2%.

2.1 Optimise with randomized search

When the number of parameter combinations because unreasonable large for grid search, and alternative is to use random search, which will select parameters randomly from the ranges given. The number of combinations tried is given by the argument `n_iter`.

Below is an example where we expand the number of arguments varied (becoming too large for grid search) and use random search to test 50 different samples.

For more details see https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.Randomiz

```
In [7]: ## Use Grid search to optimise
        # n_jobs is set to -1 to use all cores on the CPU
        from sklearn.model_selection import RandomizedSearchCV

        param_grid = {'n_estimators': [10, 30, 100, 300, 1000, 3000],
                      'bootstrap': [True, False],
                      'min_samples_split': range(2,11),
                      'max_depth': range(1,30),
                      'min_samples_split': [2, 4, 6, 8, 10],
                      'n_jobs': [-1]}

        n_iter_search = 50

        from sklearn.ensemble import RandomForestClassifier
        forest_grid = RandomForestClassifier()
        random_search = RandomizedSearchCV(forest_grid, param_grid, cv=10,
                                   n_iter=n_iter_search, scoring='accuracy')

        random_search.fit(X_train, y_train); #';' suppresses printed output

/home/michael/anaconda3/lib/python3.7/site-packages/sklearn/model_selection/_search.py:841: De
  DeprecationWarning)


In [8]: # show best parameters
        # If best parameters are at the extremes of the searches then extend the range

        random_search.best_params_

Out[8]: {'n_jobs': -1,
         'n_estimators': 100,
         'min_samples_split': 2,
         'max_depth': 29,
         'bootstrap': False}

In [9]: # Or, show full description
        random_search.best_estimator_

Out[9]: RandomForestClassifier(bootstrap=False, class_weight=None, criterion='gini',
                    max_depth=29, max_features='auto', max_leaf_nodes=None,
                    min_impurity_decrease=0.0, min_impurity_split=None,
                    min_samples_leaf=1, min_samples_split=2,
                    min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=-1,
                    oob_score=False, random_state=None, verbose=0,
                    warm_start=False)

In [10]: # Use optimised model
         model = random_search.best_estimator_
         model.fit (X_train, y_train);

In [11]: 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)
```

```
Machine learning diagnostic performance measures:
-------------------------------------------------
accuracy = 0.986
sensitivity = 0.989
specificity = 0.981
positive_likelihood = 52.411
negative_likelihood = 0.011
false_positive_rate = 0.019
false_negative_rate = 0.011
positive_predictive_value = 0.989
negative_predictive_value = 0.981
precision = 0.989
recall = 0.989
f1 = 0.989
positive_rate = 0.629
```