PennState World Campus

DAAN862: Analytics Programming in Python

Lesson 7: Evaluation Models with Scikit-Learn

Lesson 7: Objectives and Overview (1 of 6)

Lesson 7: Evaluation Models with Scikit-Learn

In Python, there are several libraries which provide a range of machine learning algorithms. The most popular one is Scikit-Learn:

Scikit-Learn: A package containing various common algorithms.

Scikit-learn features a clean, uniform, and streamlined process that has very detailed online documentation. Once you learned the basic usage and syntax for one algorithm it is very straightforward to use any classification, regression, clustering, or preprocessing models.

At the end of this lesson, using scikit-learn, the student will be able to:

Perform train-test splitting and corss validation methods with model_selection model

Evaluate classification, regression, and clustering models

By the end of this lesson, please complete all readings and assignments found in the Lesson 7 Course Schedule.

Lesson 7.1: Cross Validation (2 of 6)

Lesson 7.1: Cross Validation

In data mining, you should never use the same data to build and test models since a model would fit the data perfectly but still fail to show good performance on unseen data. The situation is referred to overfitting, and to avoid overfitting it's common to split the data into training and test datasets or us the k-fold cross validation.

Sklearn is the package name for Scikit-Learn. Since sklearn is a big package, we usually only import objects (algorithms, datasets, metrics, etc) needed instead of importing everything like we do for pandas and numpy packages.

Let's take a look at train-test split, cross validation, and cross validate function:

Train-Test Split:

Scikit-learn provides a *train test split* function which can split data into training and test sets randomly. Again, we will use iris data as the example shown in Figure 7.1. Instead of importing iris.csv, we will use the built-in iris dataset. Let's load the necessary package and datasets:

```
In [1]: import pandas as pd

In [2]: from sklearn import datasets
```

Fig 7.1 (click to enlarge)

Figure 7.2 shows the independent variables (X), dependent variables (Y), feature names, and specie categories:

```
In [4]: X,y = iris.data, iris.target

In [5]: iris.feature_names
Out[5]:
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']

In [6]: iris.target_names
Out[6]: array(['setosa', 'versicolor', 'virginica'], dtype='<U10')

In [7]: X.shape, y.shape
Out[7]: ((150, 4), (150,))
```

Fig 7.2 (click to enlarge)

Figure 7.2 shows:

Independent variables X (sepal length, sepal width, petal length and petal width).

Dependent variable y (species) are saved in iris.data and iris.target, respectively.

Feature names and specie categories are saved in iris.feature_names and iris.target_names.

The data is ready now. You can used the *train test split* function to split it into train and test set. The function can be imported from *sklearn model selction* as shown in Figure 7.3:

```
In [8]: from sklearn.model_selection import train_test_split
```

Fig 7.3 (click to enlarge)

Now you are able to split both X and y into 60% of training and 40% (shown in Figure 7.4) of test sets at the same time. The rand state is used to make sure you can repeat this example:

```
In [9]: X_train, X_test, y_train, y_test = train_test_split(
   ...:         iris.data, iris.target, test_size=0.4, random_state=0)

In [10]: X_train.shape, y_train.shape
Out[10]: ((90, 4), (90,))

In [11]: X_test.shape, y_test.shape
Out[11]: ((60, 4), (60,))
```

Fig 7.4 (click to enlarge)

The size of the training and test set can be viewed from the *shape* method.

Figure 7.5 shows an example of how to build a tree model with training data which evaluates accuracies for both training and test sets. (More of a detailed explanation will be discussed in Lesson 8):

```
In [12]: from sklearn import tree

In [13]: clf = tree.DecisionTreeClassifier().fit(X_train, y_train)

In [14]: clf.score(X_train, y_train)
Out[14]: 1.0

In [15]: clf.score(X_test, y_test)
Out[15]: 0.95
```

Fig 7.5 (click to enlarge)

Cross Validation:

K-fold cross-validation randomly splits train data into k subsets which is called folds. It choose 1 fold as test set and the rest 9 fold as train set, build a model with train data and evaluate the model on both train and test data. The process will repeat for 10 times, and each fold will take turns as test set. The simplest way to perform a k-fold cross-validation is to use *cross val score* (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html#sklearn.model_selection.cross_val_score) :

Click the dropdown to view the cross val score:

cross_val_score(*estimator*, *X*, *y=None*, *groups=None*, *scoring=None*, *cv=None*, *n_jobs=1*, *verbose=0*, *fit_params=None*, *pre_dispatch='2*n_jobs'*)

Figure 7.6 demonstrates how to estimate the accuracy of *Decision Tree Classifier* on the iris dataset by splitting the data, fitting a model and computing the score 10 consecutive times (with different splits each time). The result is an array which contains 10 accuracy scores (for classification model, the default one is accuracy):

```
In [16]: from sklearn.model_selection import cross_val_score

In [17]: clf = tree.DecisionTreeClassifier()

In [18]: accuracies = cross_val_score(clf, X_train, y_train, cv = 10)

In [19]: accuracies
Out[19]:
array([1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 0.77777778, 1.        , 0.875     , 1.        ])

In [20]: pd.Series(accuracies).describe()
Out[20]:
count    10.000000
mean      0.965278
std       0.076704
min       0.777778
25%       1.000000
50%       1.000000
75%       1.000000
max       1.000000
dtype: float64
```

Fig 7.6 (click to enlarge)

Since we didn't specify the scoring method it will use the default score method (Accuracy) of the *Decision Tree Classifier*. You can change it by using the *score* parameter which can be found in Tables 7.1.1 through 7.1.3:

Table 7.1.1: Classification Scoring Parameter

Scoring	Function	Comment

Scoring	Function	Comment
'accuracy'	metrics.accuracy score	
'average precision'	metrics.average precision score	
'f1'	metrics.f1_score	for binary targets
'f1_micro'	metrics.f1_score	micro-averaged
'f1_macro'	metrics.f1_score	macro-averaged
'f1_weighted'	metrics.f1_score	weighted average
'f1_samples'	metrics.f1_score	by multilabel sample
'neg log loss'	metrics.log loss	requires predict proba support
'precision' etc	metrics.precision score	suffixes apply as with 'f1'
'recall' etc	metrics.recall score	suffixes apply as with 'f1'
'roc auc'	metrics.roc auc score	

Table 7.1.2: Clustering Scoring Parameter

Scoring	Function
'adjusted mutual info score'	metrics.adjusted_mutual_info_score
'adjusted rand score'	metrics.adjusted_rand_score
'completeness score'	metrics.completeness_score
'fowlkes mallows score'	metrics.fowlkes_mallows_score
'homogeneity score'	metrics.homogeneity_score
'mutual info score'	metrics.mutual_info_score
'normalized mutual info score'	metrics.normalized_mutual_info_score
'v measure score'	metrics.v_measure_score

Table 7.1.3: Regression Scoring Parameter

Scoring	Function
'explained variance'	metrics.explained_variance_score
'neg mean absolute error'	metrics.mean_absolute_error
'neg_mean_squared_error'	metrics.mean_squared_error
'neg_mean_squared_log_error'	metrics.mean_squared_log_error
'neg_median_absolute_error'	metrics.median_absolute_error
'r2'	metrics.r2_score

For more details please refer to this website (http://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter) .

Cross Validate Function:

Another way to perform a k-fold cross validation is to use the *cross validate* (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_validate.html) function:

Click the dropdown to view the *cross validate* function

cross_validate(*estimator*, *X*, *y=None*, *groups=None*, *scoring=None*, *cv=None*, *n_jobs=1*, *verbose=0*, *fit_params=None*, *pre_dispatch='2*n_jobs'*, *return_train_score='warn'*)

Cross_Validate differs from coss val score in two ways:

- You can specify multiple metrics instead of just one for evaluation.
- You have an option to return a dict containing training scores, fit-times and score-times in addition to the test score.

return train score= True is set by default (as shown in Figure 7.7). If you don't need train scores, this can be set to False.

```
In [21]: from sklearn.model_selection import cross_validate

In [22]: scoring = ['accuracy', 'precision_macro', 'recall_macro']

In [23]: scores = cross_validate(clf, X_train, y_train, scoring = scoring,
    ...:                         cv = 10, return_train_score = True)

In [24]: pd.DataFrame(scores).mean()
Out[24]:
fit_time                0.000201
score_time              0.000702
test_accuracy           0.965278
train_accuracy          1.000000
test_precision_macro    0.969444
train_precision_macro   1.000000
test_recall_macro       0.961111
train_recall_macro      1.000000
dtype: float64
```

Fig 7.7 (click to enlarge)

The output *scores* is a dict. You can convert it to a DataFrame for statistical analysis.

Transcript

Started from Lesson 7, we will use scikitlearn package to build and evaluate models.

In this video, I will walk through how to perform train-test split, cross validation.

First, we import pandas and sklearn into python. Sklean is a big packge, typically we only import objects we need.

Datasets object contains several built-in datasets.

Type iris, it will show all attribute of iris: data, target, target_names, DESCR, and feature names.

We asignn iris.data and iris target to X, and y. Now you can view dimension of x and y.

Train_test_split function is the one used to split data into train and test sets. You can split X and y at the same time. Test_size is 40% of the data. And you set the random state to reproduce the result.

Now, you can view train test size to make sure everything is OK.

Here, we use a tree model as an example, more information will be provided later.

We select decision tree classifer from tree object. Fit function is used to fit the model. Without using fit function, you just create a tree object.

You can use built-in score function, which gives the accuracy. This is the accuracy for train and test sets.

Now we introduce how to perform cross validation in Python. There are two ways. The first one is using cross val score function. You need to provide the classifer or regressor, x_train, y_train, and cv, cross validation number. By default, it will use the accuracy as the score, you can change it by using scoring parameter.

This is your result.

You can perform a statistic analysis on accuracy after convert it to a Series.

The next one is called cross_validate. In this function, you can use multiple scoring metrics and could have output such as fit-time, score-time, metrics for both train and test data.

First, we import this function from sklearn and we use accuracy, precision_macro, recall_macro as the metrics. In cross_validate function, we need to give x train, y_trian, scoring, cv, and set return train score as true. The function returns a dict object, we can convert to pandas dataframe for further analysis.

Set_option is used to display all columns.

This is final result, you have fit_time, score_time, accuracy, precision, recall for both train and test

data.

You can check the mean and std of these variables.

I won't provide videos for the rest of the lesson, since these will be introduced in the later lessons.

Lesson 7.2: Classification Metrics

The sklearn.metrics module provides several score and utility functions to evaluate classification performance. In this section, we continue using Decision tree and iris data as the sample. *y pred* is the predicted value for "Species" variable as shown in Figure 7.8:

```
In [28]: clf = tree.DecisionTreeClassifier()

In [29]: clf.fit(X_train, y_train)
Out[29]:
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, 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, presort=False, random_state=None,
            splitter='best')

In [30]: y_pred = clf.predict(X_test)
```

Fig 7.8 (click to enlarge)

Accuracy, Cohen's Kappa, Confusion Matrix, Classification Report, Precission, Recall, and F1-measure:

(click tabs to learn more)

Accuracy:

The *accuracy score* (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html#sklearn.metrics.accuracy_score) function calculates the accuracy. You can choose either the fraction (default) or the count (normalize=False) of correct predictions. Figure 7.9 shows an example of how to use it:

```
In [31]: from sklearn.metrics import accuracy_score

In [32]: accuracy_score(y_test, y_pred)
Out[32]: 0.95

In [33]: accuracy_score(y_test, y_pred, normalize=False)
Out[33]: 57
```

Fig 7.9 (click to enlarge)

Cohen's Kappa:

The function *cohen kappa score* *(http://scikit-learn.org/stable/modules/generated/sklearn.metrics.cohen_kappa_score.html#sklearn.metrics.cohen_kappa_score)* calculates Cohen's kappa statistic. The range of the kappa score is between -1 and 1. Scores above 0.8 can be considered as a good model, and zero or lower meaning randomly predicted. Figure 7.10 shows an example of how ti use it:

```
In [34]: from sklearn.metrics import cohen_kappa_score

In [35]: cohen_kappa_score(y_test, y_pred)
Out[35]: 0.924114671163575
```

Fig 7.10 (click to enlarge)

Confusion Matrix:

The *confusion matrix* (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html#sklearn.metrics.confusion_matrix) function calculates the confusion matrix. Figure 7.11 shows an example of how to get a confusion matrix:

```
In [36]: from sklearn.metrics import confusion_matrix

In [37]: cm = confusion_matrix(y_test, y_pred)

In [38]: cm
Out[38]:
array([[16,  0,  0],
       [ 0, 22,  1],
       [ 0,  2, 19]], dtype=int64)
```

Fig 7.11 (click to enlarge)

In order to show the confusion matrix in a better format, you can convert it to a DataFrame and use species categories for both index and column labels as shown in Figure 7.12:

```
In [39]: pd.DataFrame(cm, index = iris.target_names, columns = iris.target_r
Out[39]:
            setosa  versicolor  virginica
setosa          16           0          0
versicolor       0          22          1
virginica        0           2         19
```

Fig 7.12 (click to enlarge)

For binary classification problems, you can extract true positives and false positives as shown in Figure 7.13:

```
In [40]: tn, fp, fn, tp = confusion_matrix([0, 1, 0, 1], [1, 1, 1, 0]).ravel(

In [41]: (tn, fp, fn, tp)
Out[41]: (0, 2, 1, 1)
```

Fig 7.13 (click to enlarge)

Classification Report:

The *classification report* (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html#sklearn.metrics.classification_report) function gives a text report of the main classification metrics as shown in Figure 7.14:

```
In [42]: from sklearn.metrics import classification_report

In [43]: print(classification_report(y_test, y_pred,
target_names=iris.target_names))
             precision    recall  f1-score   support

     setosa       1.00      1.00      1.00        16
 versicolor       0.92      0.96      0.94        23
  virginica       0.95      0.90      0.93        21

avg / total       0.95      0.95      0.95        60
```

Fig 7.14 (click to enlarge)

The report shows the precision, recall, f1-score, and support for each category and the avg/total is the average scores. The print function is used to show the results of the classification report nicely.

Precision, Recall, and F1-Measure:

The *precision score* (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html#sklearn.metrics.precision_score) , *recall score* (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html#sklearn.metrics.recall_score) , and *f1 score* (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html#sklearn.metrics.f1_score) functions are used to calculate the precision, recall and f1 scores. Figure 7.15 shows an example of how to calculate these scores:

```
In [44]: from sklearn import metrics

In [45]: metrics.precision_score(y_test, y_pred, average = 'weighted')
Out[45]: 0.9505555555555555

In [46]: metrics.recall_score(y_test, y_pred, average = 'weighted')
Out[46]: 0.95

In [47]: metrics.f1_score(y_test, y_pred, average = 'weighted')
Out[47]: 0.949922158796056
```

Fig 7.15 (click to enlarge)

The average is set to "binary" by default, which is not applicable for iris data since it has three categories. This is where we choose the weighted average. By using weighted average, the model will calculate metrics for each label and finid thier average that is weighted by support.

Tips to choose classification metrics:

1. For balanced datasets (different categories in y has the relative same frequency), you can choose accuracy as the metrics.
2. For binary classfication, you can also use ROC curve.
3. For unbalanced datasets, it is relative triky to choose a matrix.
    - The confusion matrix can be used to roughly check the model performance.
    - If you are interested in majority category, you can choose precision.
    - If you are interested in minority category, you can choose recall.
    - If you are interested in both, you can use F1 score.
    - There is a tradeoff between precision and recall. You cannot have both.

Lesson 7.3: Regression Metrics

The **sklearn.metrics** module implements several functions to measure regression performance:

mean_squared_error, *mean_absolute_error, explained_variance_score* and *r2_score*

For evaluation of regression models, we use the built-in diabetes (https://www4.stat.ncsu.edu/~boos/var.select/diabetes.html) data as the example. We can get the data information from it's attributes. Figure 7.16 shows how you can split the data into a training and test set:

```
In [4]: diabetes = datasets.load_diabetes()

In [5]: diabetes.DESCR
Out[5]: 'Diabetes dataset\n================\n\nNotes\n-----\n\nTen baseline
variables, age, sex, body mass index, average blood\npressure, and six blood serum
measurements were obtained for each of n =\n442 diabetes patients, as well as the
response of interest, a\nquantitative measure of disease progression one year
after baseline.\n\nData Set Characteristics:\n\n  :Number of Instances: 442\n\n
:Number of Attributes: First 10 columns are numeric predictive values\n\n
:Target: Column 11 is a quantitative measure of disease progression one year after
baseline\n\n  :Attributes:\n    :Age:\n    :Sex:\n    :Body mass index:\n
:Average blood pressure:\n    :S1:\n    :S2:\n    :S3:\n    :S4:\n    :S5:\n
:S6:\n\nNote: Each of these 10 feature variables have been mean centered and
scaled by the standard deviation times `n_samples` (i.e. the sum of squares of
each column totals 1).\n\nSource URL:\nhttp://www4.stat.ncsu.edu/~boos/var.select/
diabetes.html\n\nFor more information see:\nBradley Efron, Trevor Hastie, Iain
Johnstone and Robert Tibshirani (2004) "Least Angle Regression," Annals of
Statistics (with discussion), 407-499.\n(http://web.stanford.edu/~hastie/Papers/
LARS/LeastAngle_2002.pdf)\n'

In [6]: diabetes.feature_names
Out[6]: ['age', 'sex', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6']

In [7]: X, y = diabetes.data, diabetes.target

In [8]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,
   ...:                                                     random_state = 123)
```

Fig 7.16 (click to enlarge)

The linear regression model is used in Figure 7.17 since it allows you to build the model quickly and easier to interpret. A more detailed explanation will be in Lesson 9.

```
In [9]: from sklearn import linear_model

In [10]: lr = linear_model.LinearRegression()

In [11]: lr.fit(X_train, y_train)
Out[11]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
normalize=False)

In [12]: y_pred = lr.predict(X_test)
```

Fig 7.17 (click to enlarge)

y-pred is the predicted value for the test set.

Explained Variance Score, Mean Absolute Error, Mean Squared Error, and R^2 score, The Coefficient of Determination:

(click the tabs to learn more)

Explained Variance Score:

The *explained variance score* (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.explained_variance_score.html#sklearn.metrics.explained_variance_score) calculates the explained variance regression score.

If $\hat{y}$ is the predicted target output, y the actual target output, and Var is Variance, then the explained variance is calculated as follow:

$$\text{explained} - \text{variance}(y, \hat{y}) = 1 - \frac{Var(y - \hat{y})}{Var(y)}$$

The best score is 1.0, and smaller values are worse.

Example

Figure 7.18 shows an example of how to import and use *explained_variance_score*:

```
In [13]: from sklearn.metrics import explained_variance_score

In [14]: explained_variance_score(y_test, y_pred)
Out[14]: 0.5082517451456277
```

Caption 7.18 (click to enlarge)

Mean Absolute Error:

The *mean absolute error* (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html#sklearn.metrics.mean_absolute_error) function calculates mean absolute error (MAE).

The MAE equation is defined below:

$$MAE(y, \hat{y}) = \frac{1}{n_{sample}} \sum_{i=0}^{n_{sample}} |y_i - \hat{y}_i|$$

Where n_{sample} is the total number of the data.

Example

Figure 7.19 shows an example of how to use *mean absolute error*:

```
In [15]: from sklearn.metrics import mean_absolute_error

In [16]: mean_absolute_error(y_test, y_pred)
Out[16]: 44.48057319064365
```

Fig 7.19 (click to enlarge)

Mean Squared Error:

The *mean squared error* (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html#sklearn.metrics.mean_squared_error) function computes mean square error (MSE).

The MSE equation is defined below:

$$MSE(y, \hat{y}) = \frac{1}{n_{sample}} \sum_{i=0}^{n_{sample}-1} (y_i - \hat{y}_i)^2$$

Example

Figure 7.20 shows an example of *mean squared error*:

```
In [17]: from sklearn.metrics import mean_squared_error

In [18]: mean_squared_error(y_test, y_pred)
Out[18]: 2926.800577246883
```

Fig 7.20 (click to enlarge)

R^2 Score, The Coefficient of Determination:

The *r2 score* (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html#sklearn.metrics.r2_score) function computes R^2, the coefficient of determination. It shows how much data variance can be explained by the model. The best score is 1 and it is possible to get a negative value.

The R2 score equation is defined below:

$$R^2 = 1 - \frac{\sum_{i=0}^{n_{sample}-1} (y_i - \hat{y}_i)^2}{\sum_{i=0}^{n_{sample}-1} (y_i - \bar{y})^2}$$

Where $\bar{y} = \frac{1}{n_{sample}} \sum_{i=0}^{n_{sample}-1} y_i$

Example

Figure 7.21 shows an example of how to use *r2_score*:

```
In [19]: from sklearn.metrics import r2_score

In [20]: r2_score(y_test, y_pred)
Out[20]: 0.5078285584893742
```

Fig 7.21 (click to enlarge)

If you predict multiple variables instead of one, you can add the multioutput parameter in the r2_score function. The possible values for the multioutput are:

'raw_values' : A full set of scores.

'uniform_average' : Scores of different outputs are averaged with uniform weight.

'variance_weighted': Scores of different outputs are averaged, weighted by the variances of each individual output.

If the actual target value and the predicted target value are *y true* and *y pred* as listed in Figure 7.22. You can calculate the *variance weighted* R2 scores.

```
In [21]: y_true = [[0.5, 1], [-1, 1], [7, -6]]

In [22]: y_pred = [[0, 2], [-1, 2], [8, -5]]

In [23]: r2_score(y_true, y_pred, multioutput='variance_weighted')
Out[23]: 0.9382566585956417
```

Fig 7.22 (click to enlarge)

Tips to select regression metrics:

- Typically, MSE and MSE are used as the evaluation metrics for regression models.
- If there are a lot of outliers in dependent variables, it is better to use MAE since MSE penalized heavily on outliers.
- R-squared and explained variance are often used for explanatory purpose. They explain how well independent variables are selected to explain the variation of dependent variables.

Lesson 7.4: Clustering Metrics

The evaluation of clustering models is not as straightforward as the metrics for the classification problem. Instead of comparing the absolute values of true labels and assigned labels we evaluate the similarity of them.

First, let's prepare a clustering model. In preparing the clustering model we use iris data and Kmeans clustering model to assign the labels to the data: *labels true* and *labels pred* which are actual labels. Figure 7.23 shows the codes for each:

Fig 7.23

Adjusted Rand Index, Mutual Information Based Scores, Homogeneity, Completes, V-Measure, and Silhouette Coefficient

(click the tabs to learn more)

Adjusted Rand Index:

adjusted rand score *(http://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_rand_score.html#sklearn.metrics.adjusted_rand_score)* calculates the adjusted rand index. It measures the similarity of the true class labels and clustering algorithm assigned labels, which ignores the permutations of the labels.

The range of the adjusted rand index is [0, 1]. 0 means random labeling and 1 means true class labels is identical to assigned labels. In this section, we import the entire metrics package as shown in Figure 7.24, it shows that we can use metrics.functionname as a function:

Fig
7.24

Mutual Information Based Scores:

mutual info score (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_mutual_info_score.html#sklearn.metrics.adjusted_mutual_info_score) calculates the mutual information based scores. It measures the agreement of the true class labels and clustering algorithm assigned labels ignoring the permutations. In addition, there are two other versions of this measure: *Normalized Mutual Information* and *Adjusted Mutual Information* (shown in Figure 7.25):

```
In [30]: metrics.adjusted_rand_score(labels_true, labels_pred)
Out[30]: 0.7302382722834697

In [31]: metrics.mutual_info_score(labels_true, labels_pred)
Out[31]: 0.8255910976103357

In [32]: metrics.normalized_mutual_info_score(labels_true, labels_pred)
Out[32]: 0.7582057278194196
```

Fig 7.25 (click to enlarge)

Homogeneity, Completeness, and V-Measure:

Scikit-learn provided some intuitive measures suing conditional entropy analysis:

homogeneity: each cluster contains only members of a single class.

completeness: all members of a given class are assigned to the same cluster.

homogeneity score *(http://scikit-learn.org/stable/modules/generated/sklearn.metrics.homogeneity_score.html#sklearn.metrics.hom*

ogeneity_score) and *completeness score* (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.completeness_score.html#sklearn.metrics.completeness_score) are the functions to calculate these scores. Both are in the range of [0,1] (higher is better). Figure 7.26 shows an example of how to use these functions:

```
In [33]: metrics.homogeneity_score(labels_true, labels_pred)
Out[33]: 0.7514854021988339

In [34]: metrics.completeness_score(labels_true, labels_pred)
Out[34]: 0.7649861514489816
```

Fig 7.26 (click to enlarge)

The harmonic mean of homogeneity and completeness is called V-Measure, which is computed by *v measure score* (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.v_measure_score.html#sklearn.metrics.v_measure_score) . Figure 7.27 shows an example of how to use this function:

```
In [35]: metrics.v_measure_score(labels_true, labels_pred)
Out[35]: 0.7581756800057786
```

Fig 7.27 (click to enlarge)

Silhouette Coefficient:

If the actual labels for the data are unknown you cannot use the methods introduced so far to evaluate clustering results. In this case, you have to use Silhouette Coefficient as defined below:

$$s = \frac{b - a}{\max(a, b)}$$

Where:

a: The mean distance between a sample and all other points in the same class.

b: The mean distance between a sample and all other points in the *next nearest cluster*.

Example

Figure 7.28 shows an example of how to use *silhouette score*. We need to fit the data and assign labels to the data. Also, we need to provide the data, assigned labels, and distance to the function in order to calculate the score:

```
In [36]: kmeans.fit(X)
Out[36]:
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=3, n_init=10, n_jobs=1, precompute_distances='auto',
    random_state=43, tol=0.0001, verbose=0)

In [37]: labels = kmeans.labels_

In [38]: metrics.silhouette_score(X, labels, metric = 'euclidean')
Out[38]: 0.5525919445213676
```

Fig 7.28 (click to enlarge)

Tips to choose clustering metrics:

- If you don't know the number of clusters, you need to use Silhouette coefficient to select the best n.
- If use clustering models for classification purpose, you need to use adjusted rand score or mutual information based score.

Lesson 7 References (6 of 6)

Lesson 7 References

- http://scikit-learn.org/stable/model_selection.html#model-selection (http://scikit-learn.org/stable/model_selection.html#model-selection)
- http://scikit-learn.org/stable/modules/clustering.html#clustering-evaluation (http://scikit-learn.org/stable/modules/clustering.html#clustering-evaluation)
- https://jakevdp.github.io/PythonDataScienceHandbook/05.00-machine-learning.html (https://jakevdp.github.io/PythonDataScienceHandbook/05.00-machine-learning.html)