Cluster Analysis with the *sklearn* ML Framework

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Table of Contents

Return to the data set from the online clothing retailer. This data is actual data from a real online retailer, with customer names and ID's deleted from the rows of data. The only data manipulation prior to analysis is to sample from the larger data set to balance the ratio of Males and Females in the analysis and to delete samples with missing data.

Previous examples of machine learning presented supervised machine learning analyses to build a model to forecast missing Gender body type from an online order form. Here, use unsupervised learning to suggest the underlying structure regarding Gender. Without supervision, does Gender emerge from the clustering customers from other variables?

This template outlines the steps for a complete cluster analysis, with the following caveat.

Validation warning: The 340 samples in this analysis are not enough samples to split the data into training and testing data sets for meaningful cluster analysis. Not doing a proper data split keeps the analysis simpler. Without a testing data set , do *not* select a solution with many clusters as that solution likely results from overfitting.

In general, unsupervised machine learning leads to a solution evaluated for fit, just as with supervised machine learning. As always, that solution should ultimately be validated on new data.

Preliminaries

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
    Analysis on 2021-08-02 at 01:16
```

```
import os
os.getcwd()
```

```
    '/content'
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Because a cluster solution depends on the initial, somewhat arbitrary initial solution, a different solution results from each analysis. This impact is lessoned by automatically running multiple solutions, each with a different starting set of clusters, the default. Set the parameter `random_state` to an odd integer, again arbitrarily selected to reproduce these solutions.

For convenience only, here set `random_state` to a specific value referenced with two analyses so that the value only needs to be changed once to generate the same solutions.

```
start=47
```

Get and Structure Data

Read and Verify

```
d = pd.read_csv('http://web.pdx.edu/~gerbing/data/BodyMeas.csv')
#d = pd.read_csv('data/BodyMeas.csv')
d.shape
```

```
    (340, 8)
```

```
d.head()
```

	Gender	Weight	Height	Waist	Hips	Chest	Hand	Shoe
0	F	200	71	43	46	45	8.5	7.5
1	F	155	66	31	43	37	8.0	8.0
2	F	145	64	35	40	40	7.5	7.5
3	F	140	66	31	40	36	8.0	9.0
4	M	230	76	40	43	44	9.0	12.0

Create the Feature Data Structure

In general, a cluster analysis typically is based on several variables. Here, more for pedagogy, select just two features so that the result can be plotted and the clusters visualized.

```
#X = d[['Weight', 'Height', 'Waist', 'Hips', 'Chest', 'Hand', 'Shoe']]
X = d[['Hand','Height']]
X.head()
```

	Hand	Height
0	8.5	71
1	8.0	66
2	7.5	64
3	8.0	66
4	9.0	76

```
n_features = X.shape[1]
print('Number of features:', n_features)
```

```
    Number of features: 2
```

Missing data check.

```
print (d.isna().sum())
print ('\nTotal Missing:', d.isna().sum().sum())
```

```
    Gender    0
    Weight    0
    Height    0
    Waist     0
    Hips      0
    Chest     0
    Hand      0
    Shoe      0
    dtype: int64

    Total Missing: 0
```

Standardize the Variables

Because cluster analysis is based on straight-line (Euclidean) distance, the variables must be on at least approximately the same scale. Here standardize each variable to a mean of 0 and a standard deviation of 1, the z-scores. Most transformed values lie between -3 and 3 IF normally distributed.

```
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
s_scaler = preprocessing.StandardScaler()
X = s_scaler.fit_transform(X)
```

```
X = pd.DataFrame(X, columns=['Hand','Height'])
X.head()
```

	Hand	Height
0	0.064668	0.632727
1	-0.449047	-0.588199
2	-0.962762	-1.076569
3	-0.449047	-0.588199
4	0.578383	1.853653

Use `describe()` to calculate descriptive statistics that verify that the transformations correctly resulted in z-scores for each of the X variables.

```
X.describe().round(3)
```

	Hand	Height
count	340.000	340.000
mean	-0.000	0.000
std	1.001	1.001
min	-2.504	-2.297
25%	-0.706	-0.832
50%	0.065	-0.100
75%	0.578	0.877
max	2.633	2.342

The mean and standard deviation of each standardized variable are 0 and 1, respectively.

Find Optimal Number of Clusters

Each cluster analysis requires first to specify the number of clusters. A single cluster analysis cannot evaluate the optimal number of clusters to obtain the best fit. Instead, run a variety of cluster analyses, treating the number of clusters as a hyper-parameter.

Hyper-Parameter Tuning

Specify the k-means analysis with the `sklearn` function `Kmeans()`. Apply the usual `sklearn` function `fit()` to do the analysis.

However, a specific cluster analysis begins from the specified number of clusters, finding the best fit for that number of clusters. Beyond the best fit for a single model, the more general question is to understand how many clusters exist in the data. Answer that question with a hyper-parameter tuning of the number of clusters, systematically running different cluster models with differing numbers of clusters.

```
from sklearn.cluster import KMeans
from sklearn import metrics
from sklearn.metrics import pairwise_distances
from sklearn.metrics import silhouette_samples, silhouette_score
```

Set the variables needed to loop through a range of different numbers of clusters. Specify the maximum number of clusters with our own defined variable *max_nc*. Define two empty arrays, *inertia* and *silhouette* for storing the corresponding error indices for each cluster analysis at a specified number of clusters.

```
max_nc = 25
inertia = []
silhouette = []
```

For the `sklearn` function `KMeans()`, specify the number of clusters with parameter `n_clusters`. Use a `for` loop that varies the index *i* to run analyses from 2 to 25 cluster solutions systematically. All lines of indented code are the code that runs in the `for` loop.

By default, try 10 different arbitrary initial cluster solutions for each cluster analysis with the `init` parameter set to `k-means++` to specify the initial cluster seeds for each solution. To provide for a more stable, optimal solution, up this default to 100 with the parameter `n_init`. The variable *start* was previously set as an arbitrary odd number to be able to recover the solution. Or, could just set this as a constant directly.

Evaluate fit of each solution in terms of cluster inertia from created data structure *model.inertia_*, as well as the average silhouette value from *silhouette_score*. Store these fit values in the previously created *inertia* and *silhouette* arrays.

Note that a complete analysis would involve a systematic grid search over the number of clusters and the number of features.

```
for i in range(2, max_nc):
    model = KMeans(n_clusters=i, init='k-means++', n_init=100, random_state=start)
    model.fit(X)
    inertia.append(model.inertia_)
    s_score = metrics.silhouette_score(X, model.labels_, metric='euclidean')
    silhouette.append(s_score)
```

Choose the Optimal Model

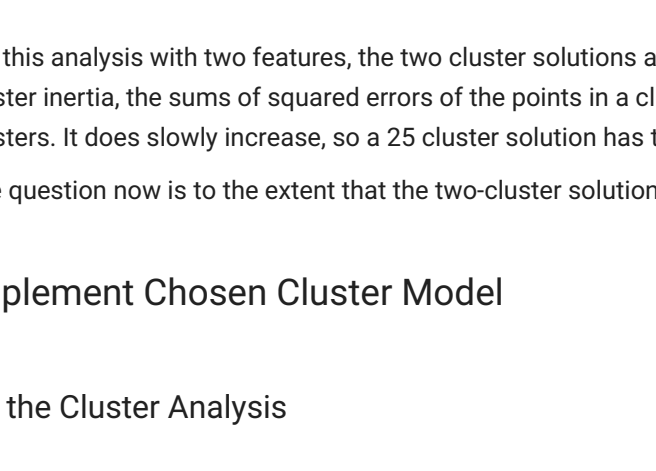
To analyze the fit of each cluster solution, display both Silhouette and Inertia for each specified number of clusters. First, print the column headings, followed by a dotted line. A `for` loop specifies the printing of the fit coefficients for each solution.

Each `format` specifies how to display the values. For the column headings, set the spacing of each. The `d` format applies to integers for the numerical output, and the `f` format applies to numbers with decimal digits, the width of the entire numerical output, and the number of decimal digits.

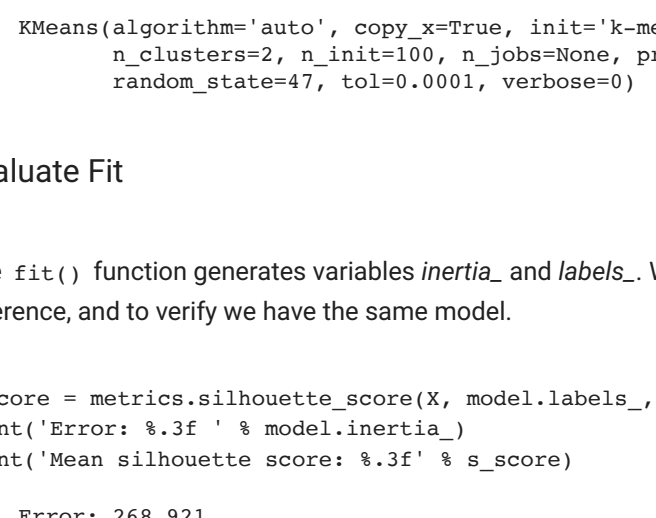
```
print('{:>2}{:>11}{:>9}'.format('nc', 'Silhouette', 'Inertia'))
print('-' * 24)
for i in range(2, max_nc):
    print('{:>2d}{:>8.3f}{:>12.3f}'.format(i, silhouette[i-2], inertia[i-2]))
```

```
    nc Silhouette  Inertia
    ------------------------
     2   0.509     268.921
     3   0.371     203.022
     4   0.351     160.368
     5   0.349     131.704
     6   0.368     110.968
     7   0.365      95.997
     8   0.386      81.539
     9   0.388      72.883
    10   0.393      65.994
    11   0.395      59.540
    12   0.403      53.671
    13   0.415      48.966
    14   0.405      45.091
    15   0.412      41.876
    16   0.402      39.236
    17   0.412      36.593
    18   0.410      33.782
    19   0.426      31.837
    20   0.447      29.498
    21   0.448      27.932
    22   0.456      25.928
    23   0.459      24.621
    24   0.469      23.060
```

```
plt.plot(range(2, max_nc), inertia, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.show()
```




```
plt.plot(range(2, max_nc), silhouette, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette')
plt.show()
```



For this analysis with two features, the two cluster solutions appear the most appropriate. As always, as the number of clusters increases, cluster inertia, the sums of squared errors of the points in a cluster about its centroid, decreases. But the silhouette score drops much after two clusters. It does slowly increase, so a 25 cluster solution has the best fit in statistical terms, but is not meaningful in interpretative terms.

The question now is to the extent that the two-cluster solution provides an interpretable solution.

Implement Chosen Cluster Model

Do the Cluster Analysis

Focus on the chosen solution of two clusters, according to the value of *n_clusters*.

```
model = KMeans(n_clusters=2, init='k-means++', n_init=100, random_state=start)
model.fit(X)
```

```
    KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
           n_clusters=2, n_init=100, n_jobs=None, precompute_distances='auto',
           random_state=47, tol=0.0001, verbose=0)
```

Evaluate Fit

The `fit()` function generates variables *inertia_* and *labels_*. We have these values from the hyper-parmater tuning, but repeat here for reference, and to verify we have the same model.

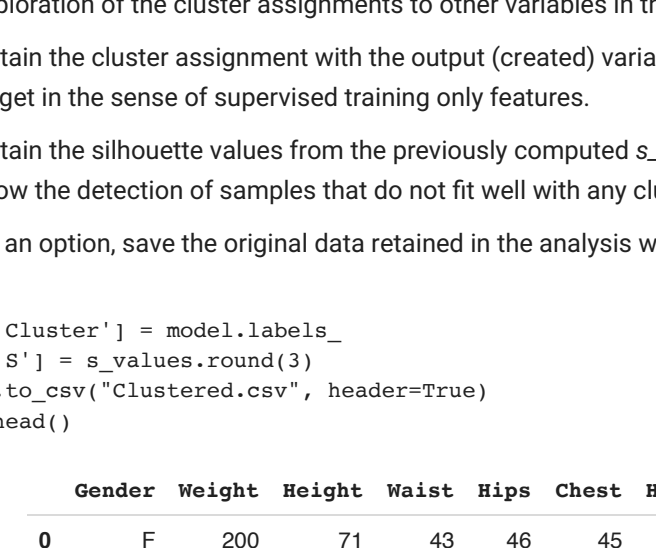
```
s_score = metrics.silhouette_score(X, model.labels_, metric='euclidean')
print('Error: %.3f ' % model.inertia_)
print('Mean silhouette score: %.3f' % s_score)
```

```
    Error: 268.921
    Mean silhouette score: 0.509
```

View the individual silhouette values from function `silhouette_samples()`. Ww want as many as possible larger than around 0.4 and 0.5.

```
s_values = silhouette_samples(X, model.labels_)
plt.hist(s_values.round(3))
plt.xlabel('Silhouette Coefficient')
plt.ylabel('Count')
```

```
    Text(0, 0.5, 'Count')
```



The two-cluster solution appears to fit the data well. Most individual silhouette are above 0.5, and there are no negative values.

Interpret the Cluster Solution

The cluster analysis has no business value unless the cluster solution can be meaningfully interpreted and related to how the world works. To do this interpretation, we need to know the cluster that each sample belongs to and the corresponding unstandardized cluster centroid (center). As usual in data analysis, the center summarizes the constituent parts, the individual rows of data, the samples, that comprise each cluster.

Assign Cluster Stats to Data

Update the original data table for each sample with the cluster assignment and the silhouette score. This enhanced data set allows further exploration of the cluster assignments to other variables in the data.

Obtain the cluster assignment with the output (created) variable *labels_*. There is no "prediction" in this context as there is no y variable as a target in the sense of supervised training only features.

Obtain the silhouette values from the previously computed *s_values* from function `silhouette_samples()`. These values are optional but allow the detection of samples that do not fit well with any cluster, should such an analysis be pursued.

As an option, save the original data retained in the analysis with `to_csv()`, though this option is commented out here with the # sign.

```
d['Cluster'] = model.labels_
d['S'] = s_values.round(3)
#d.to_csv("Clustered.csv", header=True)
d.head()
```

	Gender	Weight	Height	Waist	Hips	Chest	Hand	Shoe	Cluster	S
0	F	200	71	43	46	45	8.5	7.5	1	0.408
1	F	155	66	31	43	37	8.0	8.0	0	0.594
2	F	145	64	35	40	40	7.5	7.5	0	0.690
3	F	140	66	31	40	36	8.0	9.0	0	0.594
4	M	230	76	40	43	44	9.0	12.0	1	0.571

Cluster Centroids with Counts

Do a frequency distribution of the cluster membership. The `value_counts()` function only applies to data frame variables, so convert.

```
d_lab = pd.DataFrame(model.labels_, columns=['labels'])
count = d_lab['labels'].value_counts()
count
```

```
    0    174
    1    166
    Name: labels, dtype: int64
```

To interpret the clusters, view their centroids. View the cluster centroids as they exist in terms of the analysis for the cluster solution, that is, standardized. The created data structure `cluster_centers_` provides the centroid for each cluster, though in the metric in which the cluster analysis is conducted, here standardized.

```
dcc = pd.DataFrame(model.cluster_centers_,
          columns=['Hand', 'Height']).round(3)
dcc['Count'] = count
dcc.sort_values('Count', ascending=False)
```

	Hand	Height	Count
0	-0.736	-0.782	174
1	0.772	0.820	166

Negative standardized values indicate the number of standard deviations below the mean. Cluster 0 cutomers have below average hand size and height. The opposite is true for Cluster 1 customers.

In a cluster analysis with more than two features on which to cluster the samples, this table of the cluster centroids (centers) is the primary output for interpreting the meaning of the clusters. (With only two features, we also plot the data and cluster centroids).

To better interpret the clusters, view the cluster centers in terms of the original, unstandardized data. Compute the mean of each feature by cluster with function `groupby()` followed by the function `mean()`. Convert to a data frame and restore the variable names. Sort the clusters by frequency, from largest to smallest.

Can also save the unstandardized cluster centers to a file with the `to_csv()` function in terms of the original data as read from an external file. That option is commented out here with a #.

```
avg = d.groupby(d['Cluster']).mean().round(2)
d_avg = pd.DataFrame(avg, columns=['Hand', 'Height']).round(3)
d_avg['Count'] = count
d_avg = d_avg.sort_values('Count', ascending=False)
#d_avg.to_csv("ClustCenters.csv", header=True)
d_avg
```

	Hand	Height	Count
Cluster			
0	7.72	65.21	174
1	9.19	71.77	166

```
print('Hand size difference:', (d_avg.loc[1,'Hand'] - d_avg.loc[0,'Hand']).round(3))
print('Height difference:   ', (d_avg.loc[1,'Height'] - d_avg.loc[0,'Height']).round(3))
```

```
    Hand size difference: 1.47
    Height difference:    6.56
```

Primary interpretation: Cluster 0 samples, on average, have smaller Hand sizes and Heights, and Cluster 1 samples, on average, have larger Hand sizes and Heights.

Cluster Scatterplot

With only two features, we can also create a scatterplot to visualize the location of the samples and the cluster centroids in the plot of Hand sizes and Height.

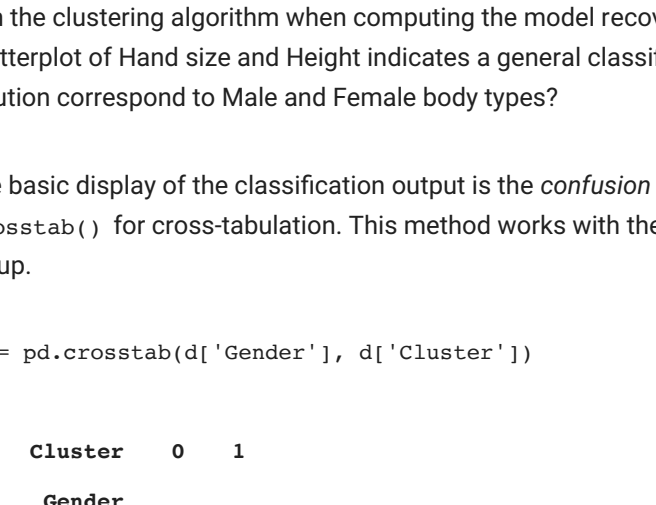
Create a `seaborn` (sns) scatterplot, with also relies upon the original `matplotlib` (plt) visualization system. The resulting scatterplot is actually two super-imposed scatterplots, one of the data and one of the cluster centroids. Parameter `s` specifies the size of the plotted points, a larger value for the centroids. Parameter `hue` sets the grouping variable. Each group is plotted in a different color according to the specified *palette*.

With the parameter `palette`, assign the colors to the points based on the `hue` specification, cluster membership. Use a Python dictionary, indicated by the { and } to set the colors. The names Cluster 0 and Cluster 1 from the cluster analysis solution are arbitrary, and, in this example, go counter to the way that the `scatterplot()` function assigns the colors to the data values. That is why the assignment of colors to the cluster centroids is reversed from the data scatterplot. The initial assignments were arbitrary, so align the centroid and data colors as needed with the dictionary.

The `if` clause indicates to run the following indented code only if *n_features* equals 2, which it does in this example. Standard Python syntax specifies that all lines of code that run when the `if` clause is true must be indented from that statement, as in the following code.

Note: If there are many samples per cluster, the cluster centroid, though displayed as a bigger circle, still blends in with the surrounding points. In that case, for printing the cluster centroids, drop the `hue` parameter, which displays a different color for each centroid, and replace with a single color such as `color='black'` or `color='DimGray'`.

```
if n_features == 2:
    sns.scatterplot(x=d['Hand'], y=d['Height'], s=50,
               hue=d['Cluster'], palette={0: 'MediumSeaGreen', 1: 'SteelBlue'})
    sns.scatterplot(x=d_avg['Hand'], y=d_avg['Height'], s=125, hue=d['Cluster'],
                   palette={1: 'MediumSeaGreen', 0: 'SteelBlue'}, legend=False)
    plt.grid()
```



With two clusters specified, the cluster solution generally recovers the Male/Female body size division based on Hand size and Height.

Evaluate Solution Against Ground Truth

After a cluster analysis, when there are other variables in the analysis, one question often of interest is if the clusters are related to other variables. Such a correspondence can suggest future supervised machine learning targeting the related variable. To uncover these relationships, do cross-tabulations of cluster membership with the values of other variables.

As an example, consider the relationship between the Cluster membership and Gender.

> *Ground truth*: The extent to which the classification of the samples reflects the true reality.

In this case, to what extent did the cluster solution, without any input from Gender, express Gender in terms of body type? Previous analyses of these data showed how supervised learning methods such as logistic regression and decision trees could predict or account for Gender. In this unsupervised analysis, there was no knowledge of Gender available to the clustering algorithm.

Can the clustering algorithm when computing the model recover the classification into groups without awareness of the groups? The previous scatterplot of Hand size and Height indicates a general classification of Male and Female. To what extent do the clusters from the two-cluster solution correspond to Male and Female body types?

The basic display of the classification output is the *confusion matrix*, previously introduced. One way to obtain is with the pandas function `crosstab()` for cross-tabulation. This method works with the non-numeric Gender values as the procudure simply counts the values in each group.

```
ct = pd.crosstab(d['Gender'], d['Cluster'])
ct
```

Cluster	0	1
Gender		
F	155	15
M	19	151

The standard procedure is for all non-numeric variables to be convered to numeric prior to machine learning. Gender was not part of this cluster analysis, and so was not converted. To proceed with `sklearn` functions in this post-analysis, we do, however, need to convert Gender to numeric.

Another way to get the confusion matrix is from the `sklearn` function `confusion_matrix()`. Its use, however, requires only numeric data, so replace the M and F data values with 1 and 0, respectively.

```
d['Gender'] = d['Gender'].replace({'M':1, 'F':0})
pd.DataFrame(metrics.confusion_matrix(d['Gender'], d['Cluster']))
```

	0	1
0	155	15
1	19	151

From the confusion matrix, obtain the traditional binary classification evaluation metrics from `sklearn`.

```
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score
print ('Accuracy: %.3f' % accuracy_score(d['Gender'], d['Cluster']))
print ('Recall: %.3f' % recall_score(d['Gender'], d['Cluster']))
print ('Precision: %.3f' % precision_score(d['Gender'], d['Cluster']))
print ('F1: %.3f' % f1_score(d['Gender'], d['Cluster']))
```

```
    Accuracy: 0.900
    Recall: 0.888
    Precision: 0.910
    F1: 0.899
```

With no knowledge of Gender, the cluster analysis obtained 90% accuracy of classification into male and female body types. This classification accuracy is particularly impressive given that it is about as accurate as the decision tree analysis, a supervised method explicitly based on the target Gender. Yet Gender did not even appear in the cluster analysis.

Plot the four values of the confusion matrix from the data structure *ct* with a two-variable bar chart.

```
ct.plot(kind='bar', color=['mediumseagreen','steelblue'])
```

```
    <matplotlib.axes._subplots.AxesSubplot at 0x7fd75639af90>
```


This cluster analysis of Hand size and Height, that is unsupervised, uncovered the structure of two distinct clusters that identify Male and Female body types. The unsupervised cluster analysis method did not include Gender, yet it was unable to cover the structure provided by the different body types.

Had our investigation of body measurements begun with the cluster analysis, by discovering a pattern, the unsupervised analysis identifies a potential target variable for future supervised analyses.

✓ 0s completed at 10:16 PM