Python Codebook

greatlearning
Learning for Life

Preface

Data Science is the art and science of solving real world problems and making data driven decisions. It involves an amalgamation of three aspects and a good data scientist has expertise in all three of them. These are:

1) Mathematical/ Statistical understanding
2) Coding/ Technology understanding
3) Domain knowledge

Your lack of expertise should not become an impediment in your journey in Data Science. With consistent effort, you can become fairly proficient in coding skills over a period of time. This Codebook is intended to help you become comfortable with the finer nuances of Python and can be used as a handy reference for anything related to data science codes throughout the program journey and beyond that.

In this document we have followed the following syntax:

- Brief description of the topic
- Followed with a code example.

Please keep in mind there is no one right way to write a code to achieve an intended outcome. There can be multiple ways of doing things in Python. The examples presented in this document use just one of the approaches to perform the analysis. Please explore by yourself different ways to perform the same thing.

Contents

Table of Figures

Unsupervised Learning

Clustering

Grouping similar data

Partition Clustering: K-Means

The KMeans algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares (see below). This algorithm requires the number of clusters to be specified. It scales well to a large number of samples and has been used across a large range of application areas in many different fields.

```
from sklearn.cluster import KMeans
import numpy as np
X = np.array([[1, 2], [1, 4], [1, 0],
              [10, 2], [10, 4], [10, 0]])
kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
kmeans.labels_ #cluster numbers assigned to data points
array([1, 1, 1, 0, 0, 0], dtype=int32) #output
kmeans.predict([[0, 0], [12, 3]])
array([1, 0], dtype=int32) #output
kmeans.cluster_centers_ #cluster centroids
array([[10.,  2.], #output
       [ 1.,  2.]])
```

Source: scikit-learn

Hierarchical Clustering: Agglomerative

Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample.

The AgglomerativeClustering object performs a hierarchical clustering using a bottom up approach: each observation starts in its own cluster, and clusters are successively merged together. The linkage criteria determine the metric used for the merge strategy.

```
from sklearn.cluster import AgglomerativeClustering
import numpy as np
X = np.array([[1, 2], [1, 4], [1, 0],
              [4, 2], [4, 4], [4, 0]])
```

```
clustering = AgglomerativeClustering().fit(X)

clustering.labels_

array([1, 1, 1, 0, 0, 0])
```

Source: scikit-learn

Dendrogram

Plotting the hierarchical clustering as a dendrogram. The dendrogram illustrates how each cluster is composed by drawing a U-shaped link between a non-singleton cluster and its children. The top of the U-link indicates a cluster merge. The two legs of the U-link indicate which clusters were merged. The length of the two legs of the U-link represents the distance between the child clusters. It is also the cophenetic distance between original observations in the two children clusters.

```
from scipy.cluster import hierarchy

import matplotlib.pyplot as plt

ytdist = np.array([662., 877., 255., 412., 996., 295., 468., 268.,

                   400., 754., 564., 138., 219., 869., 669.])

Z = hierarchy.linkage(ytdist, 'single')

plt.figure()

dn = hierarchy.dendrogram(Z)
```

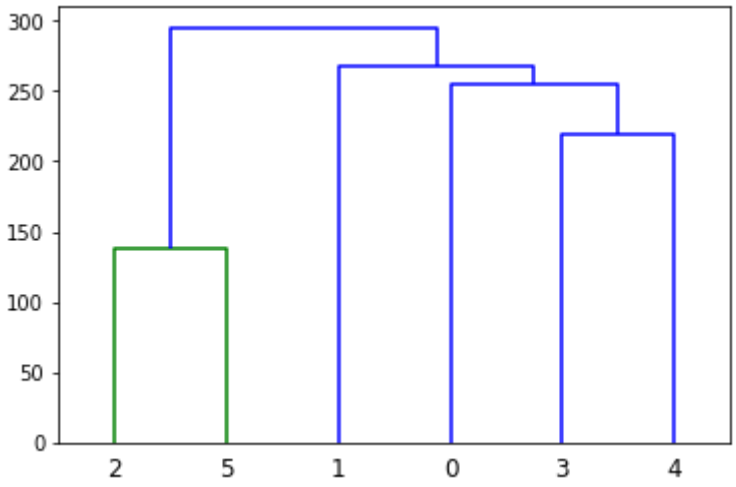

Figure 1: A Dendrogram

Source: scipy

Dimensionality Reduction Techniques

Principal Component Analysis

PCA is used to decompose a multivariate dataset in a set of successive orthogonal components that explain a maximum amount of the variance. In scikit-learn, PCA is implemented as a transformer object that learns n components in its fit method, and can be used on new data to project it on these components.

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD. source: scikit-learn

```
import numpy as np

from sklearn.decomposition import PCA

X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])

pca = PCA(n_components=2)

pca.fit(X)

PCA(n_components=2)

print(pca.explained_variance_ratio_)
```

```
[0.99244289 0.00755711]
```

```
print(pca.singular_values_)
```

```
[6.30061232 0.54980396]
```

Method 2: (using the statsmodels library)

```
from statsmodels.multivariate.pca import PCA
pc = PCA('Data Frame on which to apply PCA', ncomp='Number of principal components required')
pc.factors #to get the reduced dimension data

#Code to draw the scree plot to decide the number of factors:
from statsmodels.multivariate.factor import Factor
model=Factor('Dataset on which you want to apply PCA').fit()
model.plot_scree()
plt.show()
```

Dimensionality reduction using Linear Discriminant Analysis

LDA can be used to perform supervised dimensionality reduction, by projecting the input data to a linear subspace consisting of the directions which maximize the separation between classes. The dimension of the output is necessarily less than the number of classes, so this is, in general, a rather strong dimensionality reduction, and only makes sense in a multiclass setting.

This is implemented in **discriminant_analysis.LinearDiscriminantAnalysis.transform**. The desired dimensionality can be set using the n_components constructor parameter.

```
import numpy as np

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

X = np.array([[-1, -1,2], [-2, -1,-1], [-3, -2,-3], [1, 1,-2], [2, 1,-3], [3, 2,-2]])

y = np.array([1, 1, 1, 2, 2, 2])
```

```
lda = LinearDiscriminantAnalysis(n_components=1)
reduced=lda.fit_transform(X,y)
reduced
```

```
array([[-3.98646358],
       [-2.84747399],
       [-3.70171618],
       [ 2.27797919],
       [ 3.41696878],
       [ 4.84070578]])
```

Source: scikit-learn