kaggle Search Competitions Datasets Kernels Discussion Learn

Topic 7. Unsupervised learning: PCA and clustering

Python notebook using data from mlcourse.ai · 7,183 views · 10mo ago · beginner, clustering, learn, +1 more

93 Fork 86

Version 1
1 commit

Notebook

Topic 7. Unsupervised Learning: PCA And Clustering

Data

Log

Comments

Notebook Data Log Comments

Open Machine Learning Course

Author: Sergey Korolev (https://www.linkedin.com/in/sokorolev/), Software Engineer at Snap Inc.
Translated and edited by Egor Polusmak (https://www.linkedin.com/in/egor-polusmak/), Anastasia Manokhina (https://www.linkedin.com/in/anastasiamanokhina/), Anna Golovchenko (https://www.linkedin.com/in/anna-golovchenko-b0ba5a112/), Eugene Mashkin (https://www.linkedin.com/in/eugene-mashkin-88490883/), and Yuanyuan Pao (https://www.linkedin.com/in/yuanyuanpao/).

Topic 7. Unsupervised learning: PCA and clustering

Welcome to the seventh part of our Open Machine Learning Course!

In this lesson, we will work with unsupervised learning methods such as Principal Component Analysis (PCA) and clustering. You will learn why and how we can reduce the dimensionality of the original data and what the main approaches are for grouping similar data points.

Article outline

1. Introduction

The main feature of unsupervised learning algorithms, when compared to classification and regression methods, is that input data are unlabeled (i.e. no labels or classes given) and that the algorithm learns the structure of the data without any assistance. This creates two main differences. First, it allows us to process large amounts of data because the data does not need to be manually labeled. Second, it is difficult to evaluate the quality of an unsupervised algorithm due to the absence of an explicit goodness metric as used in supervised learning.

One of the most common tasks in unsupervised learning is dimensionality reduction. On one hand, dimensionality reduction may help with data visualization (e.g. t-SNA method) while, on the other hand, it may help deal with the multicollinearity of your data and prepare the data for a supervised learning method (e.g. decision trees).

2. Principal Component Analysis (PCA)

Intuition, theories, and application issues

Principal Component Analysis is one of the easiest, most intuitive, and most frequently used methods for dimensionality reduction, projecting data onto its orthogonal feature subspace.

More

generally speaking, all observations can be considered as an ellipsoid in a subspace of an initial feature space, and the new basis set in this subspace is aligned with the ellipsoid axes. This assumption lets us remove highly correlated features since basis set vectors are orthogonal. In the general case, the resulting ellipsoid dimensionality matches the initial space dimensionality, but the assumption that our data lies in a subspace with a smaller dimension allows us to cut off the "excessive" space with the new projection (subspace). We accomplish this in a 'greedy' fashion, sequentially selecting each of the ellipsoid axes by identifying where the dispersion is maximal.

> "To deal with hyper-planes in a 14 dimensional space, visualize a 3D space and say 'fourteen' very loudly. Everyone does it." - Geoffrey Hinton

Let's take a look at the mathematical formulation of this process:

In order to decrease the dimensionality of our data from $n$ to $k$ with $k \leq n$, we sort our list of axes in order of decreasing dispersion and take the top-$k$ of them.

We begin by computing the dispersion and the covariance of the initial features. This is usually done with the covariance matrix. According to the covariance definition, the covariance of two features is computed as follows: $$cov(X_i, X_j) = E[(X_i - \mu_i) (X_j - \mu_j)] = E[X_i X_j] - \mu_i \mu_j,$$ where $\mu_i$ is the expected value of the $i$th feature. It is worth noting that the covariance is symmetric, and the covariance of a vector with itself is equal to its dispersion.

Therefore the covariance matrix is symmetric with the dispersion of the corresponding features on the diagonal. Non-diagonal values are the covariances of the corresponding pair of features. In terms of matrices where $\mathbf{X}$ is the matrix of observations, the covariance matrix is as follows:

$$\Sigma = E[(\mathbf{X} - E[\mathbf{X}]) (\mathbf{X} - E[\mathbf{X}])^{T}]$$

Quick recap: matrices, as linear operators, have eigenvalues and eigenvectors. They are very convenient because they describe parts of our space that do not rotate and only stretch when we apply linear operators on them; eigenvectors remain in the same direction but are stretched by a corresponding eigenvalue. Formally, a matrix $M$ with eigenvector $w_i$ and eigenvalue $\lambda_i$ satisfy this equation: $M w_i = \lambda_i w_i$.

The covariance matrix for a sample $\mathbf{X}$ can be written as a product of $\mathbf{X}^{T} \mathbf{X}$. According to the Rayleigh quotient (https://en.wikipedia.org/wiki/Rayleigh_quotient), the maximum variation of our sample lies along the eigenvector of this matrix and is consistent with the maximum eigenvalue. Therefore, the principal components we aim to retain from the data are just the eigenvectors corresponding to the top-$k$ largest eigenvalues of the matrix.

The next steps are easier to digest. We multiply the matrix of our data $X$ by these components to get the projection of our data onto the orthogonal basis of the chosen components. If the number of components was smaller than the initial space dimensionality, remember that we will lose some information upon applying this transformation.

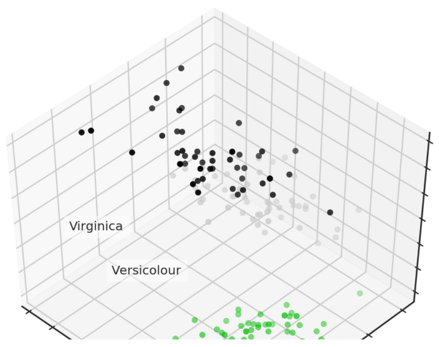
Examples

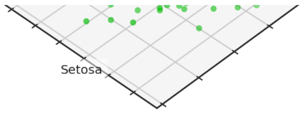
Fisher's iris dataset

Let's start by uploading all of the essential modules and try out the iris example from the `scikit-learn` documentation.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set(style='white')
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
from sklearn import decomposition
from sklearn import datasets
from mpl_toolkits.mplot3d import Axes3D

# Loading the dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Let's create a beautiful 3d-plot
fig = plt.figure(1, figsize=(6, 5))
plt.clf()
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

plt.cla()

for name, label in [('Setosa', 0), ('Versicolour', 1), (
'Virginica', 2)]:
    ax.text3D(X[y == label, 0].mean(),
              X[y == label, 1].mean() + 1.5,
              X[y == label, 2].mean(), name,
              horizontalalignment='center',
              bbox=dict(alpha=.5, edgecolor='w', facecol
or='w'))
# Change the order of labels, so that they match
y_clr = np.choose(y, [1, 2, 0]).astype(np.float)
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=y_clr,
           cmap=plt.cm.nipy_spectral)

ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([]);
```





Now let's see how PCA will improve the results of a simple model that is not able to correctly fit all of the training data:

In [2]:

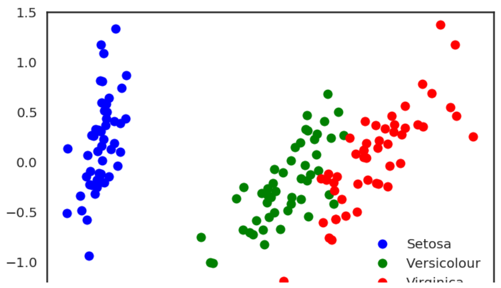
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, roc_auc_scor
e

# Train, test splits
X_train, X_test, y_train, y_test = train_test_split(X, y
, test_size=.3,
                                                    stra
tify=y,
                                                    rand
om_state=42)

# Decision trees with depth = 2
clf = DecisionTreeClassifier(max_depth=2, random_state=4
2)
clf.fit(X_train, y_train)
preds = clf.predict_proba(X_test)
print('Accuracy: {:.5f}'.format(accuracy_score(y_test,
                                                preds.ar
gmax(axis=1))))
```

```
Accuracy: 0.88889
```

Let's try this again, but, this time, let's reduce the dimensionality to 2 dimensions:

In [3]:

```
# Using PCA from sklearn PCA
pca = decomposition.PCA(n_components=2)
X_centered = X - X.mean(axis=0)
pca.fit(X_centered)
X_pca = pca.transform(X_centered)

# Plotting the results of PCA
plt.plot(X_pca[y == 0, 0], X_pca[y == 0, 1], 'bo', label
='Setosa')
plt.plot(X_pca[y == 1, 0], X_pca[y == 1, 1], 'go', label
='Versicolour')
plt.plot(X_pca[y == 2, 0], X_pca[y == 2, 1], 'ro', label
='Virginica')
plt.legend(loc=0);
```



In [4]:

```
# Test-train split and apply PCA
X_train, X_test, y_train, y_test = train_test_split(X_pc
a, y, test_size=.3,
                                                     stra
tify=y,
                                                     rand
om_state=42)

clf = DecisionTreeClassifier(max_depth=2, random_state=4
2)
clf.fit(X_train, y_train)
preds = clf.predict_proba(X_test)
print('Accuracy: {:.5f}'.format(accuracy_score(y_test,
                                                 preds.ar
gmax(axis=1))))
```

```
Accuracy: 0.91111
```

The accuracy did not increase significantly in this case, but, with other datasets with a high number of dimensions, PCA can drastically improve the accuracy of decision trees and other ensemble methods.

Now let's check out the percent of variance that can be explained by each of the selected components.

In [5]:

```
for i, component in enumerate(pca.components_):
    print("{} component: {}% of initial variance".format
(i + 1,
          round(100 * pca.explained_variance_ratio_[i],
2)))
    print(" + ".join("%.3f x %s" % (value, name)
                     for value, name in zip(component,
                                            iris.feature
_names)))
```

```
1 component: 92.46% of initial variance
0.361 x sepal length (cm) + -0.085 x sepal
width (cm) + 0.857 x petal length (cm) + 0.
358 x petal width (cm)
2 component: 5.31% of initial variance
0.657 x sepal length (cm) + 0.730 x sepal w
idth (cm) + -0.173 x petal length (cm) + -
0.075 x petal width (cm)
```

Handwritten numbers dataset

Let's look at the handwritten numbers dataset that we used before in the 3rd lesson (https://habrahabr.ru/company/ods/blog/322534/#derevya-resheniy-i-metod-blizhayshih-sosedey-v-zadache-raspoznavaniya-rukopisnyh-cifr-mnist).

In [6]:

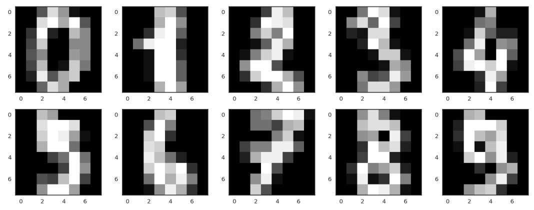
```
digits = datasets.load_digits()
```

```
X = digits.data
y = digits.target
```

Let's start by visualizing our data. Fetch the first 10 numbers. The numbers are represented by 8 x 8 matrixes with the color intensity for each pixel. Every matrix is flattened into a vector of 64 numbers, so we get the feature version of the data.

In [7]:

```
# f, axes = plt.subplots(5, 2, sharey=True, figsize=(16,
6))
plt.figure(figsize=(16, 6))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(X[i,:].reshape([8,8]), cmap='gray');
```

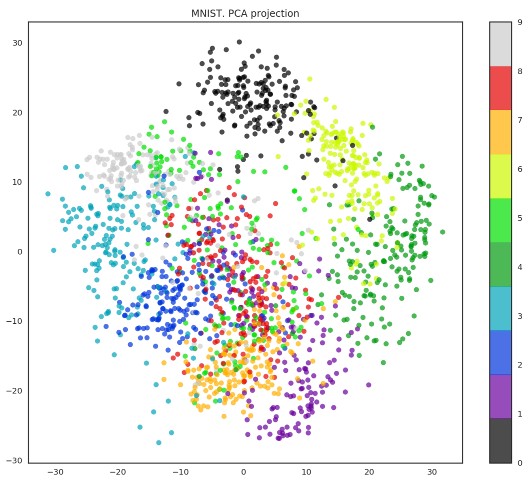


Our data has 64 dimensions, but we are going to reduce it to only 2 and see that, even with just 2 dimensions, we can clearly see that digits separate into clusters.

In [8]:

```
pca = decomposition.PCA(n_components=2)
X_reduced = pca.fit_transform(X)

print('Projecting %d-dimensional data to 2D' % X.shape[1
])

plt.figure(figsize=(12,10))
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=y,
            edgecolor='none', alpha=0.7, s=40,
            cmap=plt.cm.get_cmap('nipy_spectral', 10))
plt.colorbar()
plt.title('MNIST. PCA projection');
```

```
Projecting 64-dimensional data to 2D
```

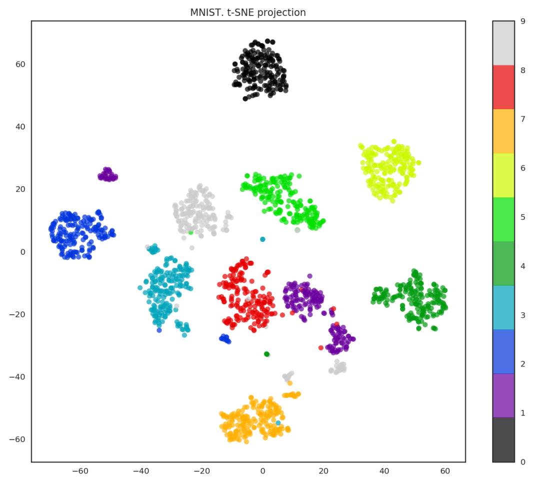


Indeed, with t-SNE, the picture looks better since PCA has a linear constraint while t-SNE does not. However, even with such a small dataset, the t-SNE algorithm takes significantly more time to complete than PCA.

In [9]:

```
%%time

from sklearn.manifold import TSNE
tsne = TSNE(random_state=17)

X_tsne = tsne.fit_transform(X)

plt.figure(figsize=(12,10))
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y,
            edgecolor='none', alpha=0.7, s=40,
            cmap=plt.cm.get_cmap('nipy_spectral', 10))
plt.colorbar()
plt.title('MNIST. t-SNE projection');
```

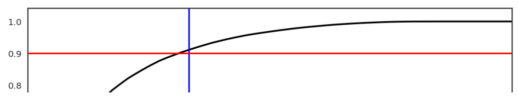
```
CPU times: user 10.5 s, sys: 48 ms, total:
10.6 s
Wall time: 10.5 s
```

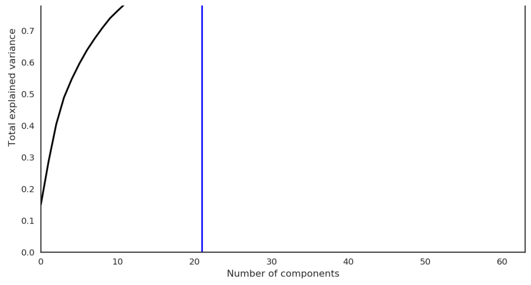


In practice, we would choose the number of principal components such that we can explain 90% of the initial data dispersion (via the `explained_variance_ratio`). Here, that means retaining 21 principal components; therefore, we reduce the dimensionality from 64 features to 21.

In [10]:

```
pca = decomposition.PCA().fit(X)

plt.figure(figsize=(10,7))
plt.plot(np.cumsum(pca.explained_variance_ratio_), color
='k', lw=2)
plt.xlabel('Number of components')
plt.ylabel('Total explained variance')
plt.xlim(0, 63)
plt.yticks(np.arange(0, 1.1, 0.1))
plt.axvline(21, c='b')
plt.axhline(0.9, c='r')
plt.show();
```





2. Clustering

The main idea behind clustering is pretty straightforward. Basically, we say to ourselves, "I have these points here, and I can see that they organize into groups. It would be nice to describe these things more concretely, and, when a new point comes in, assign it to the correct group." This general idea encourages exploration and opens up a variety of algorithms for clustering.

The examples of the outcomes from different algorithms from scikit-learn The algorithms listed below do not cover all the clustering methods out there, but they are the most commonly used ones.

K-means

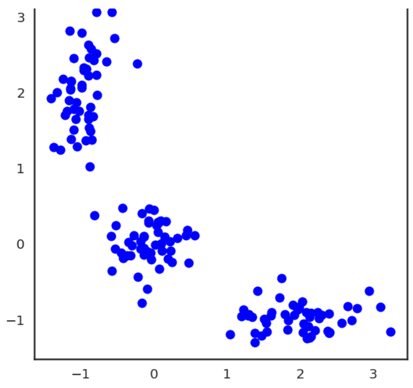
K-means algorithm is the most popular and yet simplest of all the clustering algorithms. Here is how it works:

1. Select the number of clusters $k$ that you think is the optimal number.
2. Initialize $k$ points as "centroids" randomly within the space of our data.
3. Attribute each observation to its closest centroid.
4. Update the centroids to the center of all the attributed set of observations.
5. Repeat steps 3 and 4 a fixed number of times or until all of the centroids are stable (i.e. no longer change in step 4).

This algorithm is easy to describe and visualize. Let's take a look.

In [11]:

```
# Let's begin by allocation 3 cluster's points
X = np.zeros((150, 2))

np.random.seed(seed=42)
X[:50, 0] = np.random.normal(loc=0.0, scale=.3, size=50)
X[:50, 1] = np.random.normal(loc=0.0, scale=.3, size=50)

X[50:100, 0] = np.random.normal(loc=2.0, scale=.5, size=
50)
X[50:100, 1] = np.random.normal(loc=-1.0, scale=.2, size
=50)

X[100:150, 0] = np.random.normal(loc=-1.0, scale=.2, siz
e=50)
X[100:150, 1] = np.random.normal(loc=2.0, scale=.5, size
=50)

plt.figure(figsize=(5, 5))
plt.plot(X[:, 0], X[:, 1], 'bo');
```

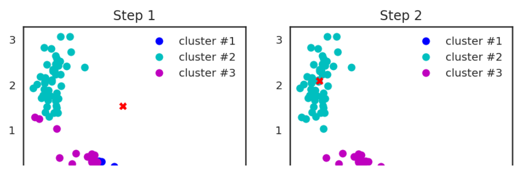


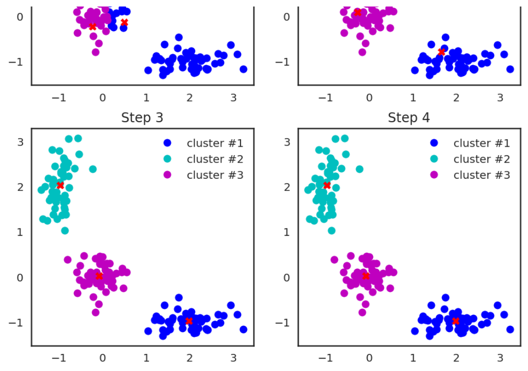
In [12]:

```
# Scipy has function that takes 2 tuples and return
# calculated distance between them
from scipy.spatial.distance import cdist

# Randomly allocate the 3 centroids
np.random.seed(seed=42)
centroids = np.random.normal(loc=0.0, scale=1., size=6)
centroids = centroids.reshape((3, 2))

cent_history = []
cent_history.append(centroids)

for i in range(3):
    # Calculating the distance from a point to a centroid
    distances = cdist(X, centroids)
    # Checking what's the closest centroid for the point
    labels = distances.argmin(axis=1)

    # Labeling the point according the point's distance
    centroids = centroids.copy()
    centroids[0, :] = np.mean(X[labels == 0, :], axis=0)
    centroids[1, :] = np.mean(X[labels == 1, :], axis=0)
    centroids[2, :] = np.mean(X[labels == 2, :], axis=0)

    cent_history.append(centroids)
```

In [13]:

```
# Let's plot K-means
plt.figure(figsize=(8, 8))
for i in range(4):
    distances = cdist(X, cent_history[i])
    labels = distances.argmin(axis=1)

    plt.subplot(2, 2, i + 1)
    plt.plot(X[labels == 0, 0], X[labels == 0, 1], 'bo',
label='cluster #1')
    plt.plot(X[labels == 1, 0], X[labels == 1, 1], 'co',
label='cluster #2')
    plt.plot(X[labels == 2, 0], X[labels == 2, 1], 'mo',
label='cluster #3')
    plt.plot(cent_history[i][:, 0], cent_history[i][:, 1
], 'rX')
    plt.legend(loc=0)
    plt.title('Step {:}'.format(i + 1));
```





Here, we used Euclidean distance, but the algorithm will converge with any other metric. You can not only vary the number of steps or the convergence criteria but also the distance measure between the points and cluster centroids.

Another "feature" of this algorithm is its sensitivity to the initial positions of the cluster centroids. You can run the algorithm several times and then average all the centroid results.

Choosing the number of clusters for K-means

In contrast to the supervised learning tasks such as classification and regression, clustering requires more effort to choose the optimization criterion. Usually, when working with k-means, we optimize the sum of squared distances between the observations and their centroids.

$$ J(C) = \sum_{k=1}^K\sum_{i~\in~C_k} ||x_i - \mu_k|| \rightarrow \min\limits_C,$$

where C – is a set of clusters with power K, μ_k is a centroid of a cluster C_k.

This definition seems reasonable -- we want our observations to be as close to their centroids as possible. But, there is a problem -- the optimum is reached when the number of centroids is equal to the number of observations, so you would end up with every single observation as its own separate cluster.

In order to avoid that case, we should choose a number of clusters after which a function $J(C_k)$ is decreasing less rapidly. More formally, $$ D(k) = \frac{|J(C_k) - J(C_{k+1})|}{|J(C_{k-1}) - J(C_k)|} \rightarrow \min\limits_k $$

Let's look at an example.

In [14]:

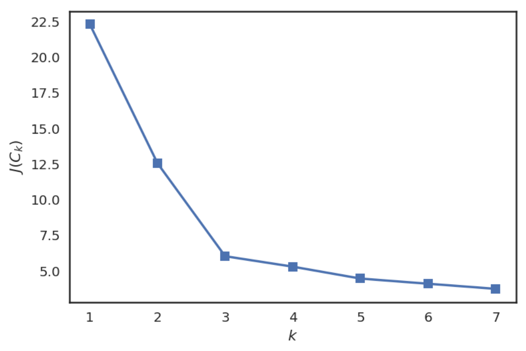
```
from sklearn.cluster import KMeans
```

In [15]:

```
inertia = []
for k in range(1, 8):
    kmeans = KMeans(n_clusters=k, random_state=1).fit(X)
    inertia.append(np.sqrt(kmeans.inertia_))
```

In [16]:

```
plt.plot(range(1, 8), inertia, marker='s');
plt.xlabel('$k$')
plt.ylabel('$J(C_k)$');
```



We see that $J(C_k)$ decreases significantly until the number of clusters is 3 and then does not change as much anymore. This means that the optimal number of clusters is 3.

Issues

Inherently, K-means is NP-hard. For $d$ dimensions, $k$ clusters, and $n$ observations, we will find a solution in $O(n^{d k+1})$ in time. There are some heuristics to deal with this; an example is MiniBatch K-means, which takes portions (batches) of data instead of fitting the whole dataset and then moves centroids by taking the average of the previous steps. Compare the implementation of K-means and MiniBatch K-means in the sckit-learn documentation (http://scikit-learn.org/stable/auto_examples/cluster/plot_mini_batch_kmeans.html).

The implemetation (http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html) of the algorithm using `scikit-learn` has its benefits such as the possibility to state the number of initializations with the `n_init` function parameter, which enables us to identify more robust centroids. Moreover, these runs can be done in parallel to decrease the computation time.

Affinity Propagation

Affinity propagation is another example of a clustering algorithm. As opposed to K-means, this approach does not require us to set the number of clusters beforehand. The main idea here is that we would like to cluster our data based on the similarity of the observations (or how they "correspond" to each other).

Let's define a similarity metric such that $s(x_i, x_j) > s(x_i, x_k)$ if an observation $x_i$ is more similar to observation $x_j$ and less similar to observation $x_k$. A simple example of such a similarity metric is a negative square of distance $s(x_i, x_j) = - ||x_i - x_j||^{2}$.

Now, let's describe "correspondence" by making two zero matrices. One of them, $r_{i,k}$, determines how well the $k$th observation is as a "role model" for the $i$th observation with respect to all other possible "role models". Another matrix, $a_{i,k}$ determines how appropriate it would be for $i$th observation to take the $k$th observation as a "role model". This may sound confusing, but it becomes more understandable with some hands-on practice.

The matrices are updated sequentially with the following rules:

$$r_{i,k} \leftarrow s_(x_i, x_k) - \max_{k' \neq k} \left\{ a_{i,k'} + s(x_i, x_k') \right\}$$

$$a_{i,k} \leftarrow \min \left(0, r_{k,k} + \sum_{i' \not\in \{i,k\}} \max(0, r_{i',k}) \right), \ \ \ i \neq k$$

$$a_{k,k} \leftarrow \sum_{i' \neq k} \max(0, r_{i',k})$$

Spectral clustering

Spectral clustering combines some of the approaches described above to create a stronger clustering method.

First of all, this algorithm requires us to define the similarity matrix for observations called the adjacency matrix. This can be done in a similar fashion as in the Affinity Propagation algorithm: $A_{i, j} = - ||x_i - x_j||^{2}$. This matrix describes a full graph with the observations as vertices and the estimated similarity value between a pair of observations as edge weights for that pair of vertices. For the metric defined above and two-dimensional observations, this is pretty intuitive - two observations are similar if the edge between them is shorter. We'd like to split up the graph into two subgraphs in such a way that each observation in each subgraph would be similar to another observation in that subgraph. Formally, this is a Normalized cuts problem; for more details, we recommend reading this paper (http://people.eecs.berkeley.edu/~malik/papers/SM-ncut.pdf).

Agglomerative clustering

The following algorithm is the simplest and easiest to understand among all the the clustering algorithms without a fixed number of clusters.

The algorithm is fairly simple:

1. We start by assigning each observation to its own cluster
2. Then sort the pairwise distances between the centers of clusters in descending order
3. Take the nearest two neigbor clusters and merge them together, and recompute the centers
4. Repeat steps 2 and 3 until all the data is merged into one cluster

The process of searching for the nearest cluster can be conducted with different methods of bounding the observations:

1. Single linkage $d(C_i, C_j) = min_{x_i \in C_i, x_j \in C_j} ||x_i - x_j||$
2. Complete linkage $d(C_i, C_j) = max_{x_i \in C_i, x_j \in C_j} ||x_i - x_j||$
3. Average linkage $d(C_i, C_j) = \frac{1}{n_i n_j} \sum_{x_i \in C_i} \sum_{x_j \in C_j} ||x_i - x_j||$
4. Centroid linkage $d(C_i, C_j) = ||\mu_i - \mu_j||$

The 3rd one is the most effective in computation time since it does not require recomputing the distances every time the clusters are merged.

The results can be visualized as a beautiful cluster tree (dendogram) to help recognize the moment the algorithm should be stopped to get optimal results. There are plenty of Python tools to build these dendograms for agglomerative clustering.

Let's consider an example with the clusters we got from K-means:

In [17]:

In [17]:

```
from scipy.cluster import hierarchy
from scipy.spatial.distance import pdist

X = np.zeros((150, 2))

np.random.seed(seed=42)
X[:50, 0] = np.random.normal(loc=0.0, scale=.3, size=50)
X[:50, 1] = np.random.normal(loc=0.0, scale=.3, size=50)

X[50:100, 0] = np.random.normal(loc=2.0, scale=.5, size=
50)
X[50:100, 1] = np.random.normal(loc=-1.0, scale=.2, size
=50)

X[100:150, 0] = np.random.normal(loc=-1.0, scale=.2, siz
e=50)
X[100:150, 1] = np.random.normal(loc=2.0, scale=.5, size
=50)

# pdist will calculate the upper triangle of the pairwise
```

Did you find this Kernel useful?
Show your appreciation with an upvote

93

Data

Data Sources

mlcourse.ai

ads.csv	216 x 2
ads_h...	2085 x 2
adult...	32.6k x 15
adult...	16.3k x 15
adult...	32.6k x 15
currenc...	300 x 2
flight_...	100k x 8
flight_...	100k x 9
hostel_fa...	18 x 12
medi...	92.2k x 3

16 more

Open Data Science

mlcourse.ai

Open Machine Learning Course by OpenDataScience

Last Updated: 10 months ago (Version 8 of 17)

About this Dataset

Open Machine Learning Course is designed to perfectly balance theory and practice; therefore, each topic is followed by an assignment with a deadline in a week. You can also take part in several Kaggle Inclass competitions held during the course and write your own tutorials. The next session starts on October 1, 2018. More info in GitHub repo.

Outline

This is the list of published articles on Medium :uk:, Habr.com :ru:, and jqr.com :cn:. Icons are clickable. 1. Exploratory Data Analysis with Pandas :uk: :ru: :cn: 2. Visual Data Analysis with Python :uk: :ru: :cn: 3. Classification, Decision Trees and k Nearest Neighbors :uk: :ru: :cn: 4. Linear Classification and Regression :uk: :ru: :cn: 5. Bagging and Random Forest :uk: :ru: :cn: 6. Feature Engineering and Feature Selection :uk: :ru: :cn: 7. Unsupervised Learning: Principal Component Analysis and Clustering :uk: :ru: 8. Vowpal Wabbit: Learning with Gigabytes of Data :uk: :ru: Kaggle Kernel 9. Time Series Analysis with Python, part 1 :uk: :ru:. Predicting future with Facebook Prophet, part 2 :uk: 10. Gradient Boosting :uk: :ru:

Assignments

Each topic is followed by an assignment. Examples are to appear in the end of June, 2018.

Kaggle competitions

1. Catch Me If You Can: Intruder Detection through Webpage

Run Info

Succeeded	True	Run Time	34.4 seconds
Exit Code	0	Queue Time	0 seconds
Docker Image Name	kaggle/python (Dockerfile)	Output Size	0
Timeout Exceeded	False	Used All Space	False
Failure Message			

Log

Download Log

```
Time   Line #  Log Message
5.4s        1  [NbConvertApp] Converting notebook script.ipynb to html
5.5s        2  [NbConvertApp] Executing notebook with kernel: python3
33.6s       3  [NbConvertApp] Support files will be in __results___files/
               [NbConvertApp] Making directory __results___files
33.7s       4  [NbConvertApp] Making directory __results___files
               [NbConvertApp] Making directory __results___files
               [NbConvertApp] Making directory __results___files
               [NbConvertApp] Making directory __results___files
               [NbConvertApp] Making directory __results___files
               [NbConvertApp] Making directory __results___files
               [NbConvertApp] Making directory __results___files
               [NbConvertApp] Making directory __results___files
               [NbConvertApp] Making directory __results___files
               [NbConvertApp] Writing 344242 bytes to __results__.html
33.7s       5
33.7s       7  Complete. Exited with code 0.
```

Comments (4)

Sort by

All Comments Hotness

Click here to comment...

Gülşah Can • Posted on Latest Version • 7 months ago • Options • Reply 1

nice!

akshatp • Posted on Latest Version • 10 months ago • Options • Reply 1

Thanks a lot!!

guoan.wen • Posted on Latest Version • 3 months ago • Options • Reply 0

I got a problem running your code, could you please help?
I ran your code in my termianl on windows, and all the plots won't appear.

When I Fork this article and ran these 3 lines of the 1st part of your code on Kaggle, it would create an empty plots. But in my terminal it creates nothing, no plot window appears, it passes on to the next steps directly:

```
fig = plt.figure(1, figsize=(6, 5))
plt.clf()
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)
```

Kernel

Yury Kash... Author • Posted on Latest Version • 3 months ago • Options • Reply 0

did you specify `%matplotlib inline` ?

Similar Kernels

Our Team Terms Privacy Contact/Support