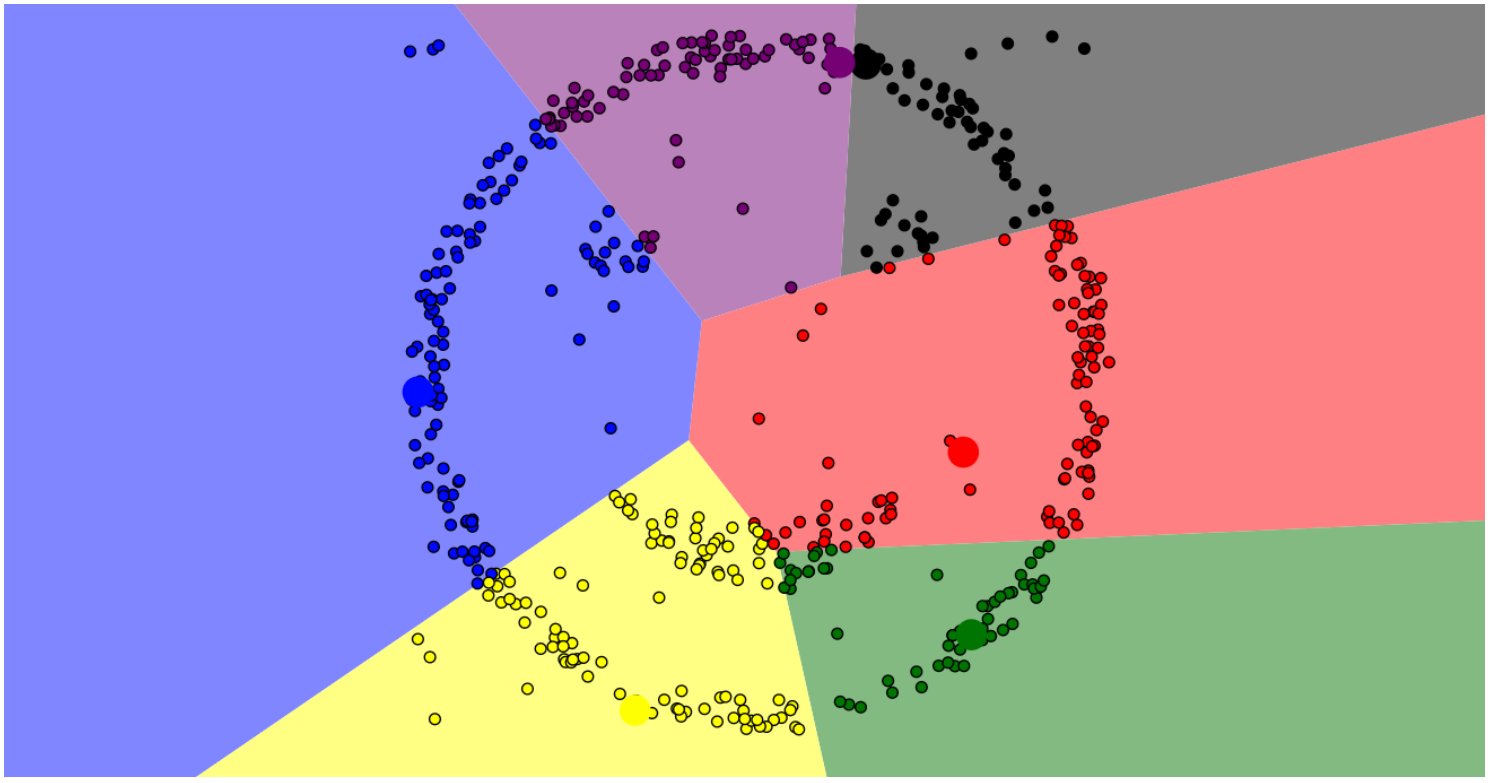


How to Build KMeans to Cluster Physical Activities on Wearable Device Dataset With Python Step-By-Step

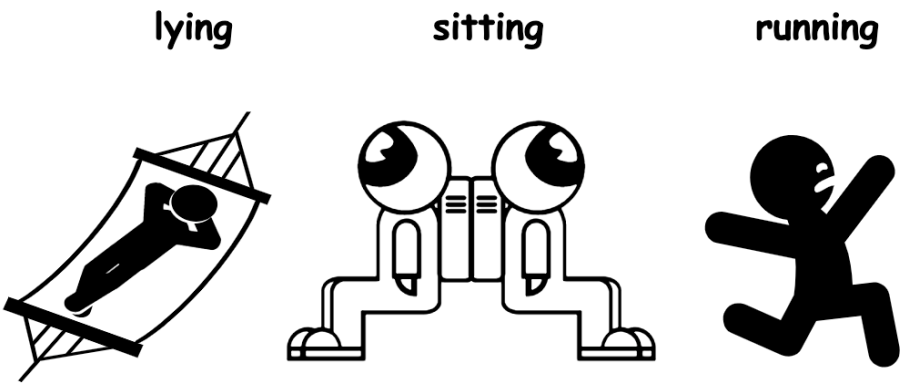
Unsupervised Learning on IoT Data using Python

Alina Zhang Follow
Feb 1 · 11 min read ★

Do you or your friend wear Fitbit, Apple Watch, or Garmin? Data collected from wearables that monitor physical activities from the wearers is being incorporated into a multitude of commercial uses including healthcare, fitness training, navigation system, and so on.

This article introduces an end-to-end unsupervised learning solution **KMeans clustering** to distinguish the physical activities of wearers: lying,

sitting or running without using any labels, from there you can develop an intuition for the end-to-end procedure of clustering with KMeans. Moreover, with the skill and experience you built with KMeans, you can create recommender systems in shopping, marketing, social media for movies, books, news, music, and so on.



icons credit to Karina M. and A M from the Noun Project

. . .

Outline

- Overall view of the data
- Statistical distribution of sensor data
- Feature scaling with standardization and one-hot encoding for activities
- Basic version of KMeans clustering in python
- Advanced version of KMeans clustering with number of clusters experiment: inertia, silhouette and overall accuracy
- Conclusion and future work

. . .

Overall view of the data

The data using in this article can be downloaded from this link which is a subset adjusted from a public dataset in UCI machine learning repository.

Import libraries

```
import os
import numpy as np
import pandas as pd
from datetime import datetime as dt
from sklearn import preprocessing as pp
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.preprocessing import label_binarize
from sklearn.cluster import KMeans, MiniBatchKMeans
from sklearn.metrics import silhouette_samples, silhouette_score
```

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style='whitegrid', palette='muted', font_scale=1.5)
```

```
pd.set_option('display.max_colwidth', -1)
pd.set_option('display.max_columns', 5000)
pd.set_option('display.max_rows' , 10000)
pd.set_option('display.width', 10000)
pd.set_option('display.precision', 6)
plt.rcParams['figure.figsize'] = [12, 6]
```

Load the data

```
df = pd.read_csv('physical activity monitoring_UCI.csv')
```

```
if 'Unnamed: 0' in df.columns.tolist():
    df = df.drop(['Unnamed: 0'], axis=1)
if 'timestamp' in df.columns.tolist():
    df = df.drop(['timestamp'], axis=1)

df.head()
```

	activity	IMUhand4	IMUhand5	IMUhand6	IMUhand7	IMUhand8	IMUhand9	IMUhand10	IMUhan
0	1	30.375	2.21530	8.27915	5.58753	2.24689	8.55387	5.77143	-0.004
1	1	30.375	2.29196	7.67288	5.74467	2.27373	8.14592	5.78739	-0.171
2	1	30.375	2.29090	7.14240	5.82342	2.26966	7.66268	5.78846	-0.238
3	1	30.375	2.21800	7.14365	5.89930	2.22177	7.25535	5.88000	-0.192
4	1	30.375	2.30106	7.25857	6.09259	2.20720	7.24042	5.95555	-0.069

```
df.info()
```

```
RangeIndex: 472576 entries, 0 to 472575
Data columns (total 52 columns):
activity      472576 non-null int64
IMUhand4      472576 non-null float64
IMUhand5      472576 non-null float64
IMUhand6      472576 non-null float64
IMUhand7      472576 non-null float64
IMUhand8      472576 non-null float64
IMUhand9      472576 non-null float64
IMUhand10     472576 non-null float64
IMUhand11     472576 non-null float64
IMUhand12     472576 non-null float64
```

```
IMUhand13     472576 non-null float64
IMUhand14     472576 non-null float64
IMUhand15     472576 non-null float64
```

As shown above, there are 472576 entries with 52 columns. The 52 features are organized as follows:

- **"activity"**

```
'''
3 types of activities
- 1 lying
- 2 sitting
- 5 running
'''

df['activity'].value_counts()
```

```
1    192290
2    184645
5    95641
Name: activity, dtype: int64
```

There are 3 types of activities in the dataset. "1" corresponds to "lying", "2" to "sitting", and "5" to "running". To make it simple, let's update the dataset by using "3" to represent "running". **This label is only used for evaluating our KMeans model rather than training the model.**

```
#  use 3 for running

conditions = [df['activity']==1,df['activity']==2,df['activity']==5]
choices = [1,2,3]
df['activity']  = np.select(conditions, choices, default='null')

df['activity'].value_counts()
```

```
1    192290
2    184645
3    95641
Name: activity, dtype: int64
```

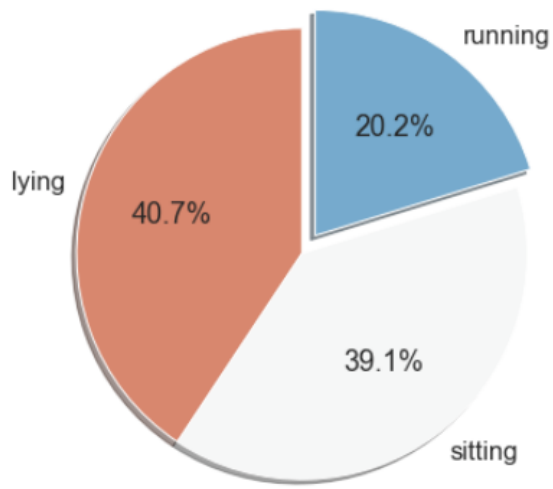
1 for lying, 2 for sitting, 3 for running

```
# Pie chart, where the slices will be ordered and plotted counter-
clockwise:
labels = 'lying', 'sitting', 'running'
sizes = [192290, 184645, 95641]
```

```
# only "explode" the 3nd slice (i.e. 'running')
explode = (0, 0, 0.1)


fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as
a circle.


plt.show()
```



20.2% of the activities are "running"

- **"IMU*" data from 3 sensors**

3 Colibri wireless IMUs (inertial measurement units) were used. The positions of the sensors and collected data:

1. 1 IMU over the wrist on the dominant **arm** — collected data stored in "IMUhand4" ~ "IMUhand20"
2. 1 IMU on the **chest** — collected data stored in "IMUchest21" ~ "IMUchest37"
3. 1 IMU on the dominant side's **ankle** — collected data stored in "IMUankle38" ~ "IMUankle54"

photo edited from a post at ebuyer

. . .

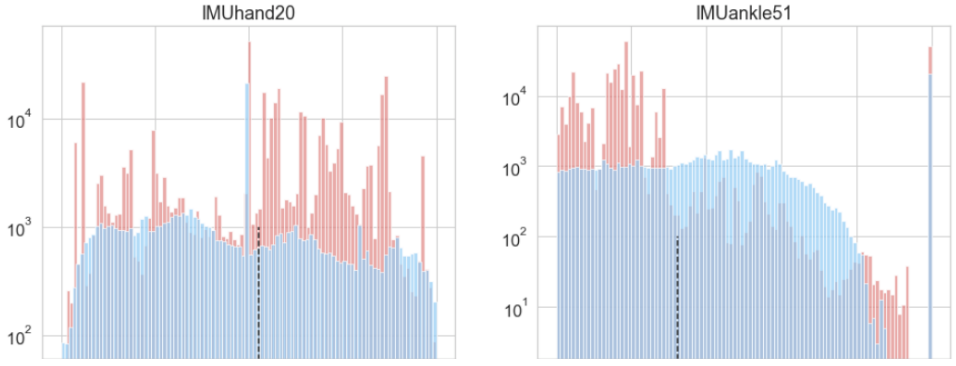
Statistical distribution of sensor data

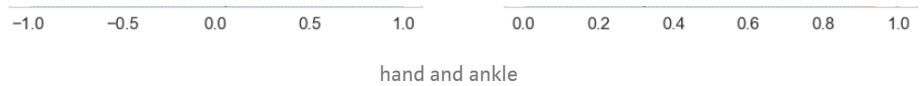
Visualize and compare the distributions of sensor data collected from three positions: hand, ankle, and chest.

```
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

axes[0].hist(df[df['activity']!=3]['IMUhand20'], bins=100, alpha=0.7,
color='LightCoral');
axes[0].hist(df[df['activity']==3]['IMUhand20'], bins=100, alpha=0.7,
color='LightSkyBlue');
axes[0].vlines(x=df['IMUhand20'].mean(),ymin=0, ymax=10**3,
linestyles='dashed')
axes[0].set_yscale('log')
axes[0].set_title('IMUhand20')

axes[1].hist(df[df['activity']!=3]['IMUankle51'], bins=100,
alpha=0.7, color='LightCoral');
axes[1].hist(df[df['activity']==3]['IMUankle51'], bins=100,
alpha=0.7, color='LightSkyBlue');
axes[1].vlines(x=df['IMUankle51'].mean(),ymin=0, ymax=10**2,
linestyles='dashed')
axes[1].set_yscale('log')
axes[1].set_title('IMUankle51');
```

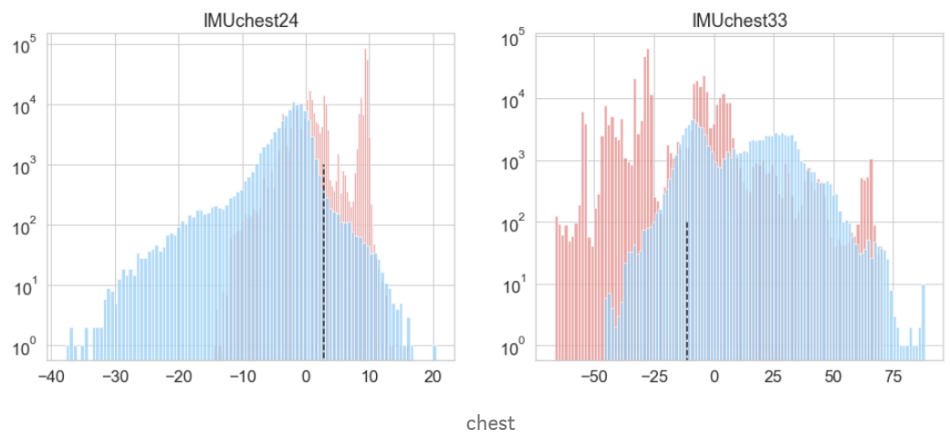





```
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

axes[0].hist(df[df['activity']!=3]['IMUchest24'], bins=100,
alpha=0.7, color='LightCoral');
axes[0].hist(df[df['activity']==3]['IMUchest24'], bins=100,
alpha=0.7, color='LightSkyBlue');
axes[0].vlines(x=df['IMUchest24'].mean(),ymin=0, ymax=10**3,
linestyles='dashed')
axes[0].set_yscale('log')
axes[0].set_title('IMUchest24')

axes[1].hist(df[df['activity']!=3]['IMUchest33'], bins=100,
alpha=0.7, color='LightCoral');
axes[1].hist(df[df['activity']==3]['IMUchest33'], bins=100,
alpha=0.7, color='LightSkyBlue');
axes[1].vlines(x=df['IMUchest33'].mean(),ymin=0, ymax=10**2,
linestyles='dashed')
axes[1].set_yscale('log')
axes[1].set_title('IMUchest33');
```



The red bars represent the wearer in “running” state while the blue bars are in either “lying” or “sitting”. We randomly picked columns: ‘IMUhand20’, ‘IMUankle51’, ‘IMUchest24’ and ‘IMUchest33’ which were data collected from hand, ankle, chest respectively.

As illustrated in the above figures, we can tell that the distributions of chest data varies more from running to non-running state comparing to hand and ankle data. It makes sense that the heart rate would go higher when running which is a strong signal.

. . .

Feature scaling with standardization

Apply a standard scaler from sklearn to standardize the features except "activity" so that these features would be centred around 0 with a standard deviation of 1. As emphasized many times in my previous posts, standardizing data to make sure the features are in the same relative range is an essential step and also a general requirement for many machine learning algorithms.

```
# standardization
data_std = data.copy().drop(['activity'], axis=1)
Class = data['activity'].copy()
featuresToScale = data_std.columns
scaler = pp.StandardScaler(copy=True, with_mean=True, with_std=True)
data_std.loc[:, featuresToScale] =
scaler.fit_transform(data_std[featuresToScale])

data_std.describe()
```

	IMUhand4	IMUhand5	IMUhand6	IMUhand7	IMUhand8	IMUhand9	IM
count	2.362900e+04	2.362900e+04	2.362900e+04	2.362900e+04	2.362900e+04	2.362900e+04	2.362
mean	6.248711e-16	-2.525946e-17	-7.216990e-18	-1.744106e-17	-8.419822e-18	3.728778e-17	-2.22
std	1.000021e+00	1.000021e+00	1.000021e+00	1.000021e+00	1.000021e+00	1.000021e+00	1.000
min	-2.671650e+00	-1.102481e+01	-6.885704e+00	-2.419989e+01	-7.993683e+00	-6.928608e+00	-1.538
25%	-6.843014e-01	-2.680057e-01	-4.208721e-01	-4.506949e-01	-2.530325e-01	-4.468830e-01	-4.52
50%	1.829053e-01	8.495135e-02	2.527680e-02	3.517622e-01	8.335456e-02	3.908539e-02	3.65
75%	7.249095e-01	7.288325e-01	3.179466e-01	5.995756e-01	7.284658e-01	3.420189e-01	6.03
max	1.447582e+00	2.798607e+00	1.320314e+01	5.534641e+00	2.836362e+00	6.160469e+00	4.744

```
# One-hot encoding

from sklearn.preprocessing import label_binarize

'''
3 activities
- 1 lying
- 2 sitting
- 3 running
'''

# one-hot encoding
n_classes = 3
lb = pp.LabelBinarizer()
y_label = lb.fit_transform(Class)
```

```
Class
```

```
121958    3
146867    1
131932    1
```

```
365838    3
259178    2
          ..
364130    3
313142    1
198462    1
293702    3
265129    2
Name: activity, Length: 23629, dtype: int32
```

```
y_label
```

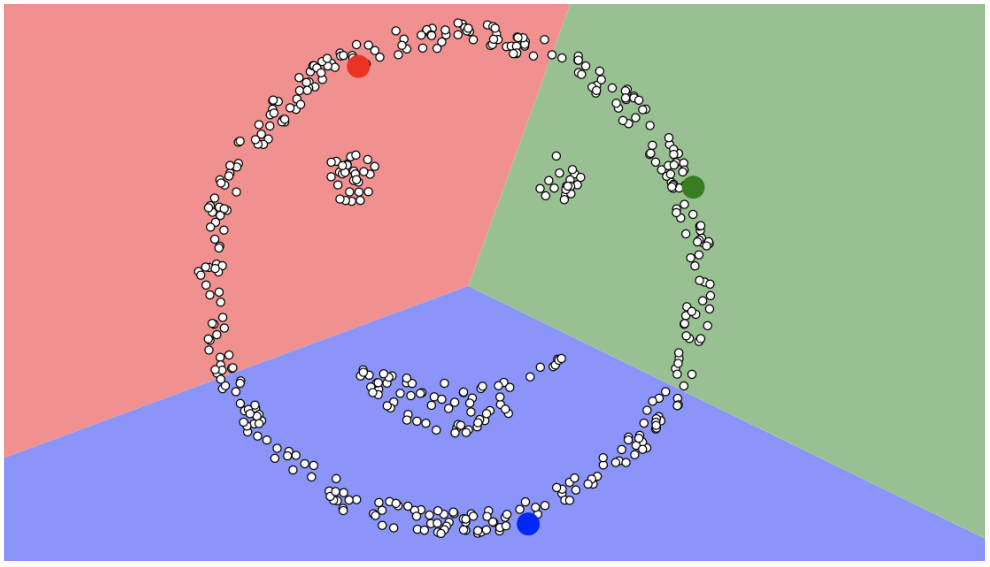
```
array([[0, 0, 1],
       [1, 0, 0],
       [1, 0, 0],
       ...,
       [1, 0, 0],
       [0, 0, 1],
       [0, 1, 0]])
```

one-hot encoding for Class which is the type of activities

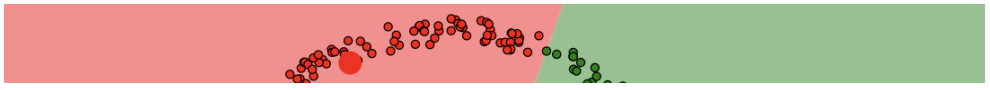
. . .

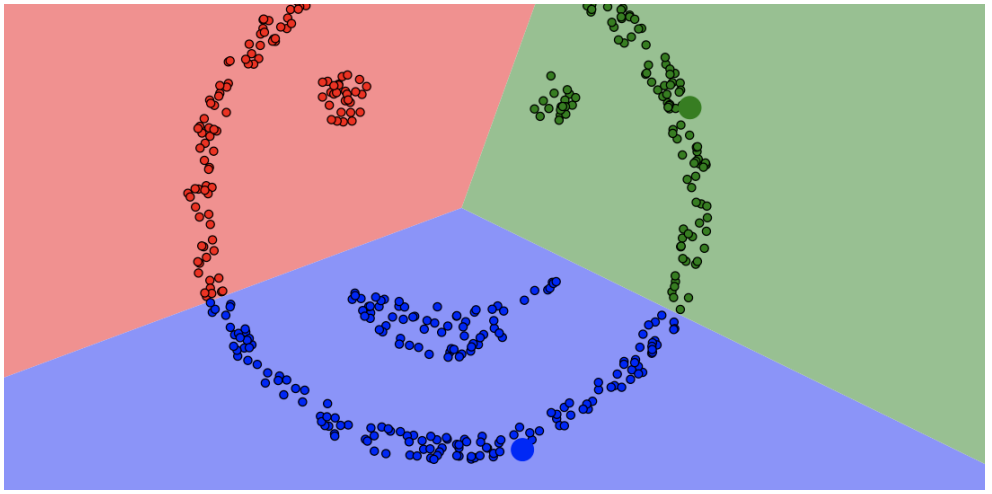
Basic version of KMeans clustering in python

Clustering is an important member in unsupervised learning family which attempts to group objects together based on similarity without using any labels. As mentioned at the beginning of the article, clustering is commonly applied to build the foundation of recommender systems in retail, online shopping, marketing, social media for news, movies, books, music, and so on.



Before clustering; 3 centroids added





After clustered with 3 centroids; Image created using an online tool

In KMeans clustering, the number of desired clusters needs to be specified before training the model. The results of different runs of KMeans would be slightly different because the algorithm randomly assigns each observation to one of the clusters to kick off the clustering, then optimizes the segmentations by minimizing the sum of within-cluster variation which is also called inertia across all clusters. Typically, KMeans does several runs and chooses the run that has the lowest sum of inertia across all clusters.

1. split the data into train and test dataset.

```
X_train , X_test , y_train , y_test = \
            train_test_split(data_std, y_label, test_size=0.2,
random_state=42, stratify=y_label)
```

2. load the libraries and set up the parameters.

- **n_init** is the number of initializations KMeans will be run with different number of clusters. The final results will be the best output of n_init consecutive runs in terms of inertia (within-cluster variation).
- **max_iter** is the maximum number of iterations of the algorithm for a single run. The total iterations would be (225 iterations/per run) * 12 runs in our case.
- **n_jobs** = -1 means we are using all processors. KMeans is really SLOW (no complaining :).

3. build and train the model; get inertia value; compute average silhouette_score.

```python
# set up the parameters
n_init = 12
max_iter = 225
tol = 0.0001
random_state = 42
n_jobs = -1
n_clusters = 3

t0 = dt.now()
print("=========  Start training ... ")

# Initialize the clusterer with n_clusters value and a random generator
# seed of 42 for reproducibility.
clusterer = KMeans(n_clusters=n_clusters, random_state=random_state)
cluster_labels = clusterer.fit_predict(X_train)

# inertia
inertia = clusterer.inertia_

# The silhouette_score gives the average value for all the samples.
# This gives a perspective into the density and separation of the formed
# clusters
silhouette_avg = silhouette_score(X_train, cluster_labels)
# Compute the silhouette scores for each sample
sample_silhouette_values = silhouette_samples(X, cluster_labels)

print("For n_clusters =", n_clusters,
      ", the inertia is :", inertia,
      ", the average silhouette_score is :", silhouette_avg)

t1 = dt.now()-t0
print("=========  Finished in ",t1)
```

basicVersionOfKMeans hosted with ♡ by GitHub view raw

Basic version of KMeans clustering python code

```
=========  Start training ...
For n_clusters = 3 , the inertia is : 735341.4988453572
 the average silhouette_score is : 0.21326449066285105
=========  Finished in  0:00:08.603051
```

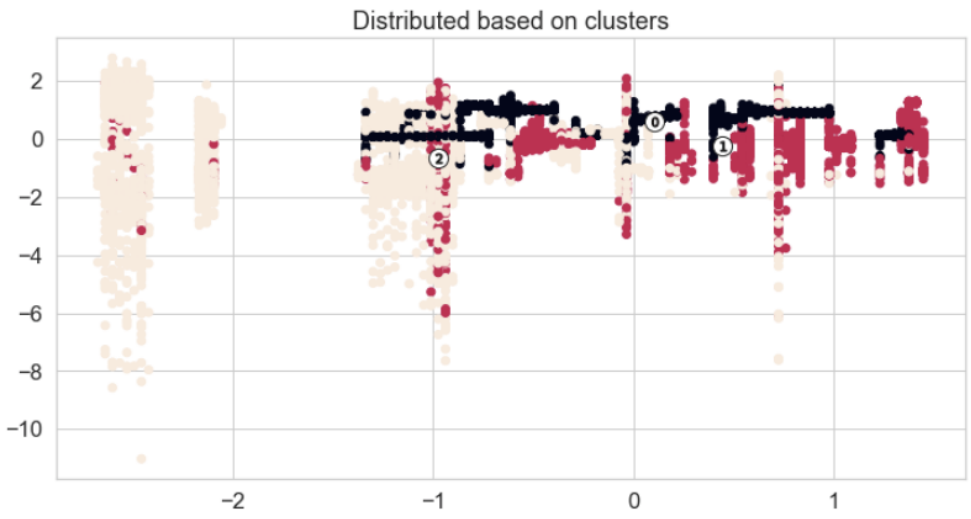
When the num of clusters is 3, the KMeans group all the observations into 3 clusters with average silhouette score 0.21 and inertia 735341.49.

```python
# plot the distribution of the 3 clusters

plt.scatter(X_train.iloc[:, 0], X_train.iloc[:, 1], c=cluster_labels,
label=cluster_labels)
# Labeling the clusters
centers = clusterer.cluster_centers_
# Draw white circles at cluster centers
plt.scatter(centers[:, 0], centers[:, 1], marker='o',
            c="white", alpha=1, s=200, edgecolor='k')

for i, c in enumerate(centers):
    plt.scatter(c[0], c[1], marker='$%d$' % i, alpha=1,
                s=50, edgecolor='k')
```

```
plt.title("Distributed based on clusters");
```



You can pause here and come back in a few days if you feel a bit overwhelmed in information, especially if it is your first to build clustering model. Too much input could be too chunky to eat with one bite. If you are super curious about how to experiment and choose the ideal number of clusters by computing inertia, silhouette score*, and overall accuracy, let us continue.*

. . .

Advanced version of KMeans clustering with number of clusters experiment: inertia, silhouette and overall accuracy

There are three metrics wildly used to evaluate the KMeans algorithm:

- Inertia
- Silhouette score
- Overall accuracy

The python code of KMeans advanced version is upgraded from the basic one with following changes:

```
# set up the parameters
n_init = 12
max_iter = 225
tol = 0.0001
random_state = 42
```

```
n_jobs = -1

t0 = dt.now()
print("=========  Start training ... ")

inertia_df = pd.DataFrame(data=[], index=range(2, 21), columns=['inertia'])
silhouette_avg_df = pd.DataFrame(data=[], index=range(2, 21), columns=['silhouetteAvg'])
overallAccuracy_df = pd.DataFrame(data=[], index=range(2, 21), columns=['overallAccuracy'])

for n_clusters in range(2, 21):
    clusterer = KMeans(n_clusters=n_clusters, n_init=n_init, max_iter=max_iter, tol = tol, \
                       random_state=random_state, n_jobs=n_jobs)
    cluster_labels = clusterer.fit_predict(X_train)

    # inertia
    inertia_df.loc[n_clusters] = clusterer.inertia_

    # The silhouette_score gives the average value for all the samples.
    # This gives a perspective into the density and separation of the formed
    # clusters
    silhouette_avg_df.loc[n_clusters] = silhouette_score(X_train, cluster_labels)
    # Compute the silhouette scores for each sample
    sample_silhouette_values = silhouette_samples(X_train, cluster_labels)

    # self-defined accuracy function: overallAccuracy
    kmeansClustered = pd.DataFrame(data=cluster_labels, index=X_train.index, columns=['cluster'
    countByCluster_kMeans, countByLabel_kMeans, \
    countMostFreq_kMeans, \
    accuracyDF_kMeans, overallAccuracy_kMeans, \
    accuracyByLabel_kMeans = overallAccuracy(kmeansClustered, pd.Series(y_Class, index=X_train.
    overallAccuracy_df.loc[n_clusters] = overallAccuracy_kMeans



    # Plot the silhouette scores for each 'n_clusters'
    # Create a subplot with 1 row and 2 columns
    fig, (ax1, ax2) = plt.subplots(1, 2)
    fig.set_size_inches(18, 7)

    # The 1st subplot is the silhouette plot
    # The silhouette coefficient can range from -1, 1 but in this example all
    # lie within [-0.1, 1]
    ax1.set_xlim([-0.1, 1])
    # The (n_clusters+1)*10 is for inserting blank space between silhouette
    # plots of individual clusters, to demarcate them clearly.
    ax1.set_ylim([0, len(X_train) + (n_clusters + 1) * 10])

    y_lower = 10
    for i in range(n_clusters):
        # Aggregate the silhouette scores for samples belonging to
        # cluster i, and sort them
        ith_cluster_silhouette_values = \
            sample_silhouette_values[cluster_labels == i]

        ith_cluster_silhouette_values.sort()

        size_cluster_i = ith_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_cluster_i

        color = cm.nipy_spectral(float(i) / n_clusters)
        ax1.fill_betweenx(np.arange(y_lower, y_upper),
                          0, ith_cluster_silhouette_values,
```

```python
                          facecolor=color, edgecolor=color, alpha=0.7)

        # Label the silhouette plots with their cluster numbers at the middle
        ax1.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))

        # Compute the new y_lower for next plot
        y_lower = y_upper + 10  # 10 for the 0 samples

    ax1.set_title("The silhouette plot for the various clusters.")
    ax1.set_xlabel("The silhouette coefficient values")
    ax1.set_ylabel("Cluster label")

    # The vertical line for average silhouette score of all the values
    ax1.axvline(x=silhouette_avg, color="red", linestyle="--")

    ax1.set_yticks([])  # Clear the yaxis labels / ticks
    ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])

    # 2nd Plot showing the actual clusters formed
    colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)
    ax2.scatter(X_train.iloc[:, 0], X_train.iloc[:, 1], marker='.', s=30, lw=0, alpha=0.7,
                c=colors, edgecolor='k')


    # Labeling the clusters
    centers = clusterer.cluster_centers_
    # Draw white circles at cluster centers
    ax2.scatter(centers[:, 0], centers[:, 1], marker='o',
                c="white", alpha=1, s=200, edgecolor='k')

    for i, c in enumerate(centers):
        ax2.scatter(c[0], c[1], marker='$%d$' % i, alpha=1,
                    s=50, edgecolor='k')

    ax2.set_title("The visualization of the clustered data.")
    ax2.set_xlabel("Feature space for the 1st feature")
    ax2.set_ylabel("Feature space for the 2nd feature")

    plt.suptitle(("Silhouette analysis for KMeans clustering on sample data "
                  "with n_clusters = %d" % n_clusters),
                 fontsize=14, fontweight='bold')



t1 = dt.now()-t0
print("=========  Finished in ",t1)

plt.show()
```

advancedKMeans.py hosted with ♡ by GitHub view raw

The python code of KMeans advanced version

- run KMeans at different number of clusters which ranges from 2 to 20
- record the inertia
- record the silhouette score

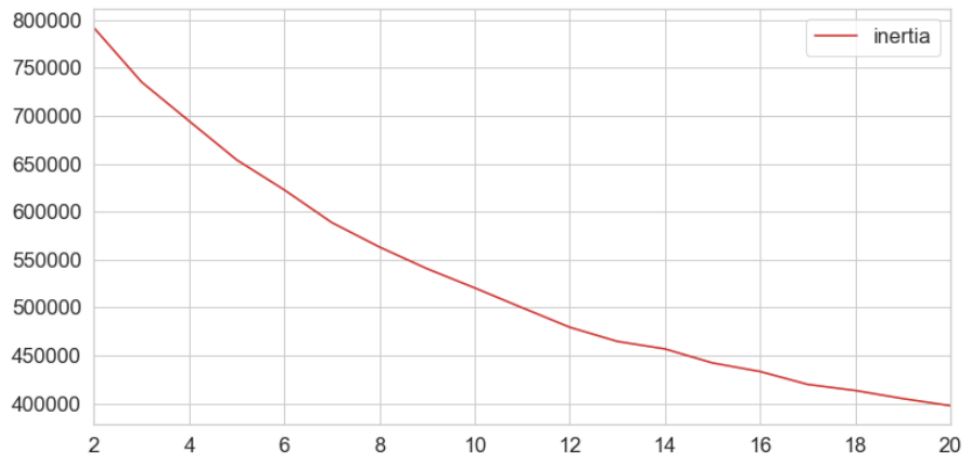
- add a self-defined function to compute overall accuracy which we will get into details below

Inertia

Inertia is the sum of squared distances of observations to their closest cluster centre which is also called within-cluster variation. The plot below shows the inertia value when number of clusters going from 2 to 20.

The inertia decreases as the number of clusters increases. This does make sense because the more clusters we have, the greater the homogeneity among observations within each cluster.

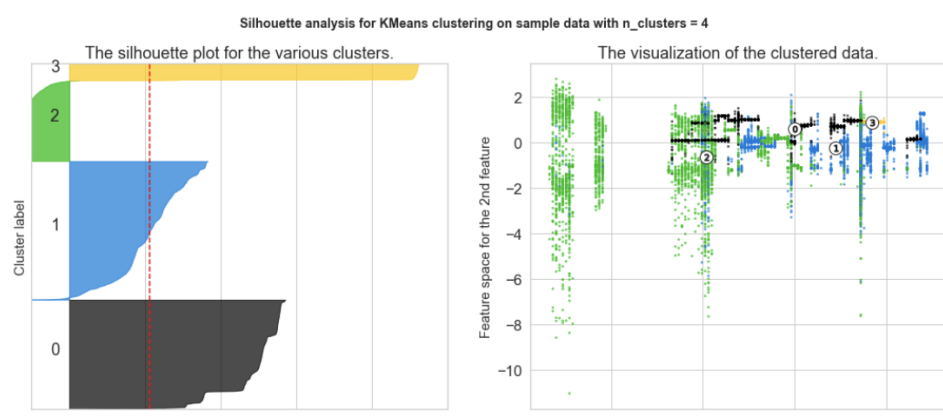
```
inertia_df.plot();
```



Silhouette score

The silhouette score can be used to study the density and separation between the formed clusters which has a range of [-1, 1].

The following silhouette plots with *n_clusters=4* displays how close each point in one cluster is to points in the neighbouring clusters:



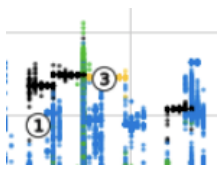
Separation between clusters

- **0<silhouette score≤1**: silhouette score close to +1 indicates that the point is far away from the neighbouring clusters and the algorithm is confident with this decision
- **silhouette score≈0**: silhouette score almost equal to 0 indicates that the point is on or very close to the decision boundary between two neighbouring clusters
- **-1≤silhouette score<0**: silhouette score with negative value indicates that those point might have been assigned to the wrong cluster

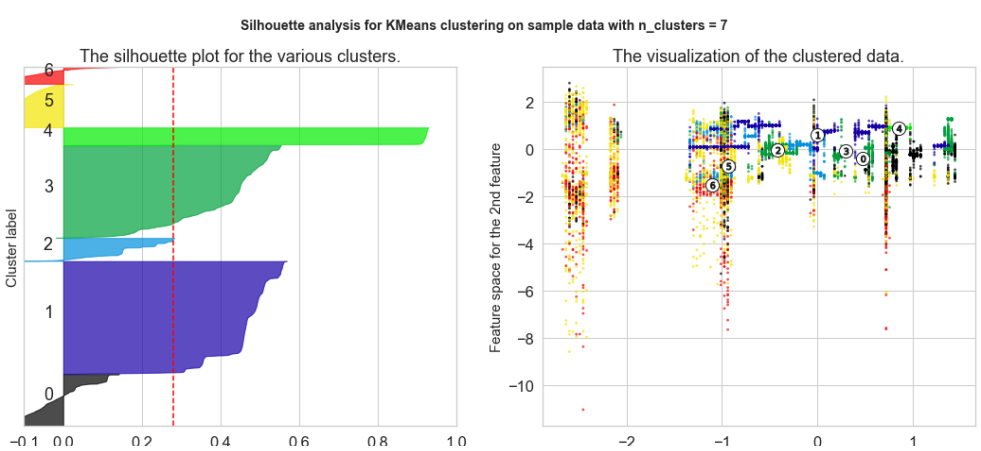
Thickness — the size of cluster

The thickness of the silhouette plot represents the cluster size. For example, ***cluster 0*** in black and ***cluster 1*** in blue are of similar thickness and hence are of similar sizes which also can be verified from the labelled scatter plot on the right (black points vs blue points).

Cluster 3 in yellow is the thinnest means that it is the smallest cluster. From the labelled scatter plot we can see that there are very few yellow points.

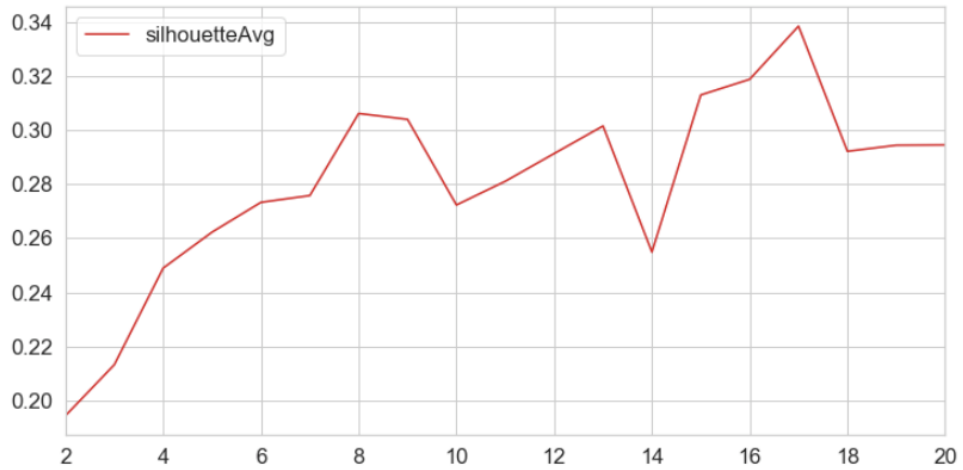


Try to analysis the silhouette plot with *n_clusters=7* below using the method we just went through.



The line plot illustrates how the silhouette score varies when n_clusters growing from 2 to 20. We can see that *n_clusters=17* could be a good candidate because the highest average silhouette score of all observations.

```
silhouette_avg_df.plot();
```



Overall accuracy

The following code defined a function *overallAccuracy(clusterDF, labelsDF)* to analyze the accuracy of each cluster and the overall accuracy.

```
def overallAccuracy(clusterDF, labelsDF):
    countByCluster = pd.DataFrame(data=clusterDF['cluster'].value_counts())
    countByCluster.reset_index(inplace=True, drop=False)
    countByCluster.columns = ['cluster', 'clusterCount']
    # print('countByCluster \n', countByCluster)

    preds = pd.concat([labelsDF, clusterDF], axis=1)
    preds.columns = ['trueLabel', 'cluster']
    # print('preds \n', preds)

    countByLabel = pd.DataFrame(data=preds.groupby('trueLabel').count())
    print('countByLabel \n', countByLabel)

    '''
    lambda x: x.value_counts().iloc[0])
    will return the most freq true label for each cluster
    i.e. for cluster 0, the true label would be 1
         for cluster 1, the true label would be 2


            cluster          trueLabel
        0       1            20739
                3            18
                2            10
        1       2            19923
```

```
                3            2857
                1            350
        2       3            8732
                2            2219
                1            1860

    '''
    countMostFreqLabel = pd.DataFrame(data=preds.groupby('cluster').agg( \
                                            {lambda x: x.value_counts().tolist()[0], \
                                             lambda x: x.value_counts().keys().tolist()[0]})
    countMostFreqLabel.reset_index(inplace=True, drop=False)
    countMostFreqLabel.columns = ['cluster', 'countMostFreqLabel','lable']
    print('countMostFreqLabel \n', countMostFreqLabel,'\n \n \n')


    accuracyDF = countMostFreqLabel.merge(countByCluster, left_on="cluster", right_on="cluster")
    print('accuracyDF: i.e. dots clustered as A, how many of them have real label A \n', accurac


    overallAccuracy = accuracyDF.countMostFreqLabel.sum() / accuracyDF.clusterCount.sum()
    print('overallAccuracy \n', overallAccuracy)


    accuracyByLabel = accuracyDF.countMostFreqLabel / accuracyDF.clusterCount
    print('accuracyByLabel \n', accuracyByLabel, '\n==================================\n \n \n'


    return countByCluster, countByLabel, countMostFreqLabel, accuracyDF, overallAccuracy, accura
```

overallAccuracy.py hosted with ♡ by GitHub view raw

Let's get into some details.

	cluster	countMostFreqLabel	label	clusterCount
0	0	5964	1	5972
1	1	6518	2	7220
2	2	931	1	934
3	3	1768	3	2345
4	4	1463	3	2432

Above screenshot is a part of the print by function *overallAccuracy*.

There are 5 clusters in total and we will count the most frequent label in each cluster and use it as clustered label. For example, in ***cluster 0***, there are 5972 observations in total. 5964 out of them have label 1 which represents ***lying*** in our case so that we will label ***cluster 0*** as ***lying***.

Likewise for other clusters. 7220 observations were grouped as ***cluster 1*** in which 6518 out of them have label 2 (***sitting***). ***Cluster 1*** will be labeled as ***sitting*** which is its most frequent label.

Different clusters can have the same label. As shown above, the most frequent labels in ***cluster 3*** and ***cluster 4*** are both 3 (***running***). The observations in these two clusters will be labeled as ***running***.

```
    cluster  countMostFreqLabel  label  clusterCount
0  0        5964                1      5972
1  1        6518                2      7220
2  2        931                 1      934
3  3        1768                3      2345
4  4        1463                3      2432
overallAccuracy
 0.8804951594984923
accuracyByLabel
 0    0.998660
1    0.902770
2    0.996788
3    0.753945
4    0.601562
dtype: float64
==================================
```

Equation to calculate accuracyByLabel and overallAccuracy

$$A: \text{ the most frequent label in the cluster}$$
$$k: \text{ there are k clusters in total}$$

$$accuracyByLabel = \frac{num\ of\ observations\ in\ the\ cluster\ having\ label\ A}{total\ num\ of\ observations\ in\ the\ cluster}$$

$$overallAccuracy = \frac{\sum_i num\ of\ observations\ in\ the\ cluster\ having\ label\ A}{\sum_i total\ num\ of\ observations\ in\ the\ cluster}, \ i = 1, 2 \ldots k$$

How to calculate accuracyByLabel for each cluster:

For ***cluster 0***, accuracy is 5964/5972=0.998660.

For ***cluster 4***, accuracy is 1463/2432=0.601562.

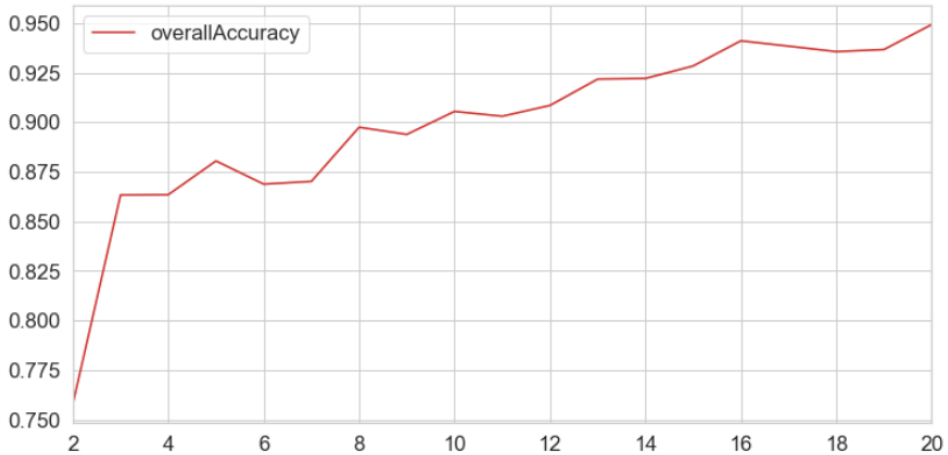
How to calculate overallAccuracy for all clusters:

To calculate the overallAccuracy, it is

$$\frac{5964 + 6518 + 931 + 1768 + 1463}{5972 + 7220 + 934 + 2345 + 2432} = 0.880495$$

The following line plot shows the overallAccuracy when n_clusters in range [2, 20].

```
overallAccuracy_df.plot();
```



Create a score table

Create a score table to save the inertia, silhouetteAvg, and overallAccuracy calculated above for different number of clusters.

```
evaluation_df = inertia_df.copy()
evaluation_df['inertia'] = inertia_df['inertia']
evaluation_df['silhouetteAvg'] = silhouette_avg_df['silhouetteAvg']
evaluation_df['overallAccuracy'] =
overallAccuracy_df['overallAccuracy']
evaluation_df['numOfClusters'] = inertia_df.index

cols = ['numOfClusters','inertia','silhouetteAvg','overallAccuracy']
evaluation_df = evaluation_df[cols]
evaluation_df
```

	numOfClusters	inertia	silhouetteAvg	overallAccuracy
2	2	791847	0.194399	0.757869
3	3	735342	0.213248	0.863302
4	4	694558	0.248957	0.863461
5	5	654023	0.262249	0.880495
6	6	622716	0.273254	0.868804
7	7	588560	0.275757	0.870179
8	8	563074	0.306101	0.897477
9	9	540448	0.303913	0.893879
10	10	520423	0.272276	0.905465
11	11	499703	0.281038	0.903031
12	12	479328	0.291288	0.90848
13	13	464535	0.301477	0.921758
14	14	456714	0.254884	0.922076
15	15	442045	0.312938	0.928371
16	16	433119	0.318713	0.941068
17	17	419670	0.338366	0.93837
18	18	413283	0.292116	0.935619

19	19	404871	0.294361	0.936677
20	20	397391	0.294453	0.949267

experiment results

Based on the experiment results, **17** could be the optimal pick for number of clusters on this dataset. In practice, we can use *n_clusters=17* to train KMeans on this dataset and use the trained model to cluster the unseen data.

. . .

Conclusion and future work

In this article, firstly, we analyzed a sensor dataset which collected from wearers in three physical activities: lying, sitting or running. Then we went through two versions of KMeans implementation with inertia, silhouette and overall accuracy introduced.

For future, what we can do to push the project into next level? At least, there are two crucial things worth considering if you are working on a real world project rather than self-learning:

- PCA powered dimensionality reduction before clustering
- MiniBatchKMeans

PCA powered dimensionality reduction before clustering

There are 51 columns of sensor data fed into the KMeans model in our example. What if we implement a dimensionality reduction using normal PCA before we run the clustering algorithm?

In my experiment, when the number of principal components equal to 5 which captured 55% variance of original dataset, the KMeans model can achieve a slightly better overallAccuracy.

Isn't it impressive? You can use **only 10%** (from 51 to 5) of the features to achieve a better performance! The reason is that dimensionality reduction can densely capture salient information in the original dataset while removing redundant information, and one of the most typical linear dimensionality reduction method is normal PCA.

I would highly recommend to add normal PCA as a part of your machine learning pipeline before clustering because it helps

- solve large-scale, computationally expensive problems
- identify interesting underlying patterns in data more effectively because a lot of the noise (redundant information) has been reduced

MiniBatchKMeans

KMeans is super super super slow. It could take forever if you run on your laptop. For large scale datasets with more than 10k observations, you can either train yourself to be more patient or use MiniBatchKMeans which is much faster implementation because of its incremental updates using mini-batches.

. . .

Wow!!! It is literally a big chunk and you are doing great if you get here. You have come a long way. Now, can you celebrate by clustering what I am doing:

Data Driven Investor

Gain Access to Expert Views

Email

First Name

Give me access!

I agree to leave Medium.com and submit this information, which will be collected and used according to Upscribe's privacy policy.

3423 signups

Data Science Machine Learning Python Artificial Intelligence IoT

Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade

About Help Legal