# Parallel Computing on Python

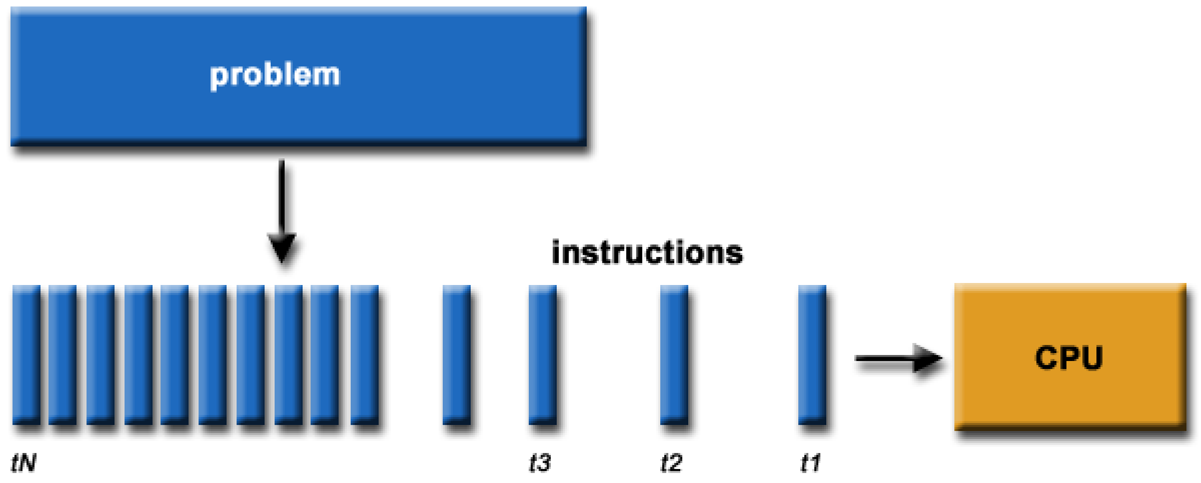
## Outline

- Parallel Computing
- K-means Clustering
- Scikit-learn package on Python
- Joblib Package
- Multiprocessing (Optional)

## Parallel Computing

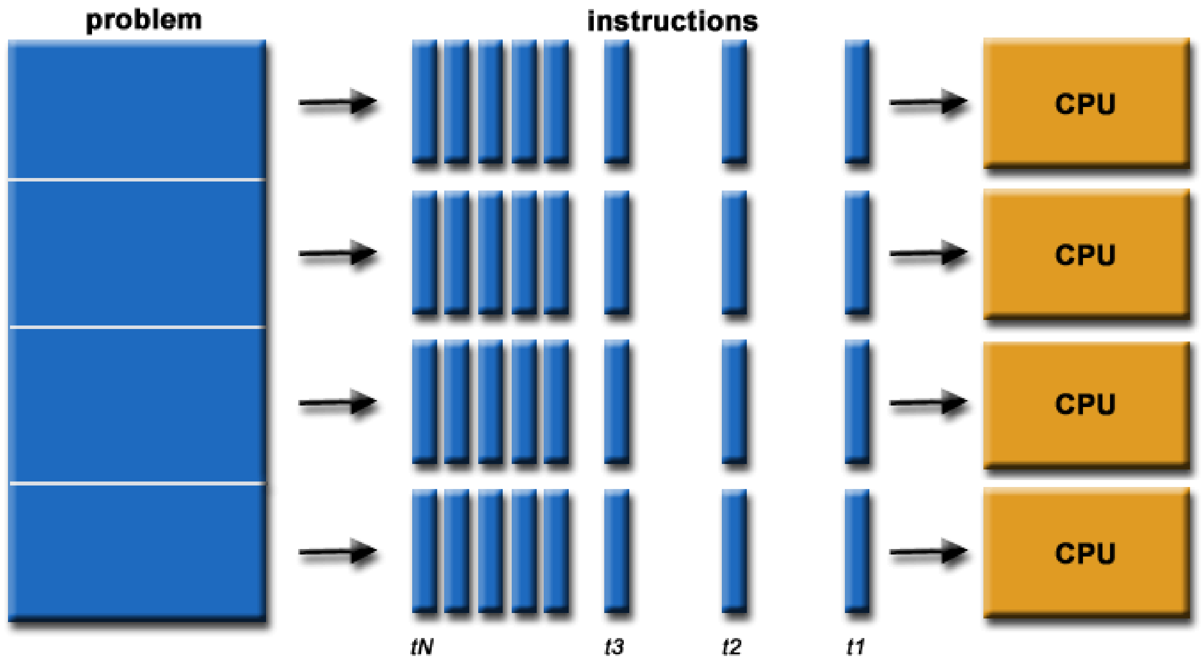
## Serial Computing

- Traditionally, software has been written for *serial* computation:
    - To be run on a single computer having a single Central Processing Unit(CPU);
    - A problem is broken into a discrete series of instructions.
    - Instructions are executed one after another.
    - Only one instruction may execute at any moment in time.



# Parallel Computing

- In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem.
  - To be run using multiple CPUs.
  - A problem is broken into discrete parts that can be solved concurrently.
  - Each part is further broken down to a series of instructions.
  - Instructions from each part execute simultaneously on a different CPUs.



http://cosy.univ-reims.fr/~fnolot/Download/Cours/HPC/introduction_to_parallel_computing.ppt

## Parallel Computing

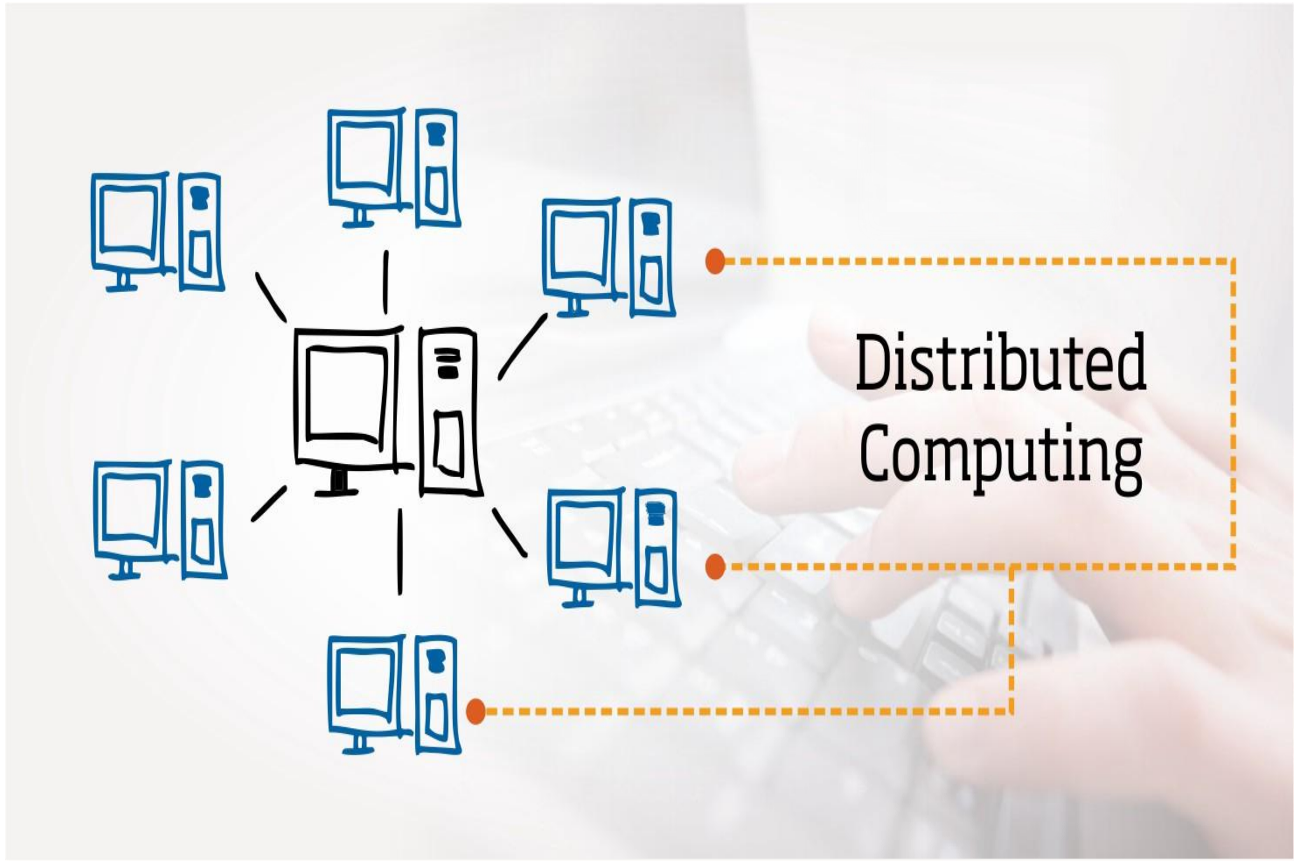
Today, commercial applications are providing an equal or greater driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways. Example applications include:

- parallel databases, data mining
- oil exploration
- web search engines, web based business services
- computer-aided diagnosis in medicine
- management of national and multi-national corporations
- advanced graphics and virtual reality, particularly in the entertainment industry
- networked video and multi-media technologies
- collaborative work environments
- Ultimately, parallel computing is an attempt to maximize the infinite but seemingly scarce commodity called time.

# Distributed Computing

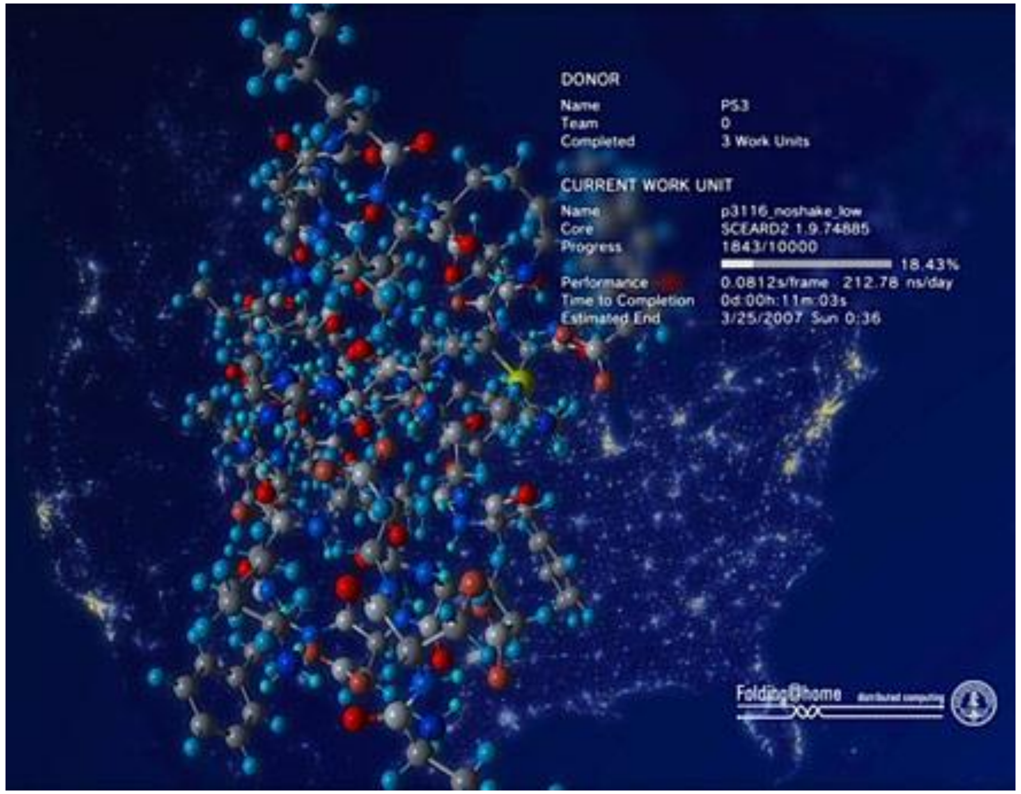
Distributed computing is a field of computer science that studies distributed systems. A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers, which communicate with each other via message passing.

--WEKIPEDIA



https://medium.com/@etraining/distributed-computing-processing-models-a5825f12329e

# Distributed Computing



DONOR

Name PS3
Team 0
Completed 3 Work Units

CURRENT WORK UNIT

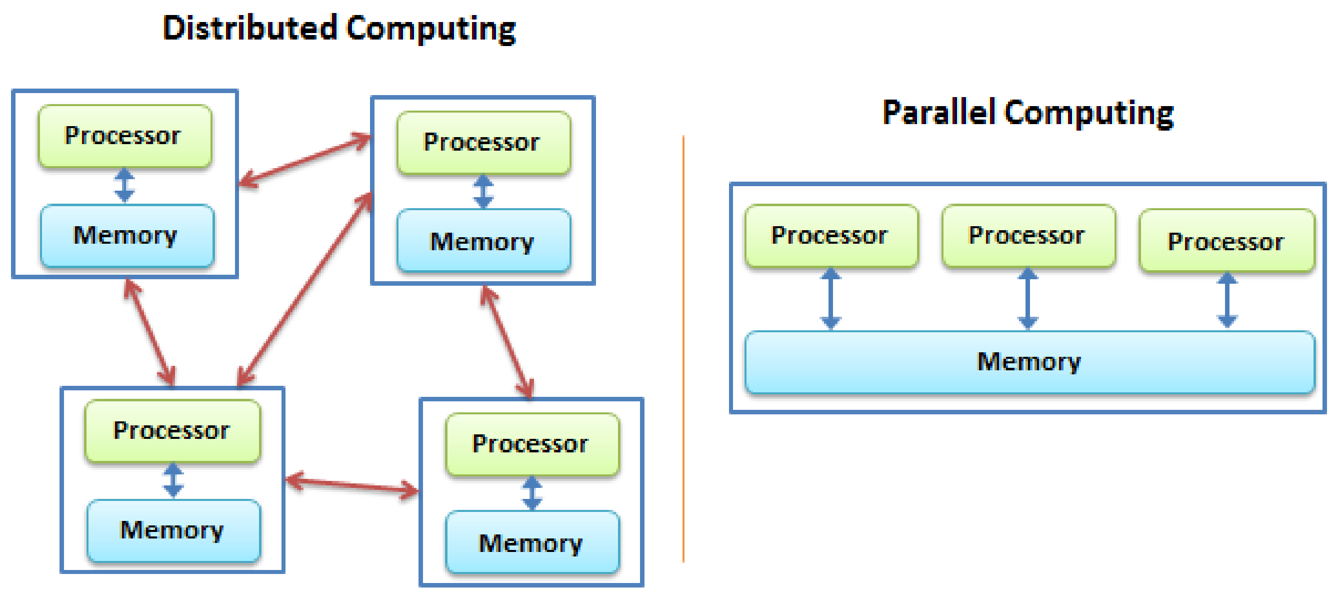
Name p3116_noshake_low
Core SCEARD2 1.9.74885
Progress 1843/10000
18.43%
Performance 0.0812s/frame 212.78 ns/day
Time to Completion 0d:00h:11m:03s
Estimated End 3/25/2007 Sun 0:36

**Folding@home** project, which analyzes the internal structure of proteins and related drugs, has a huge structure and requires an incredible amount of calculation, which is impossible to calculate by a computer. Although there are supercomputers with superb computing power, these devices are expensive, and some research institutions are very limited in funding. With distributed computing, it can be achieved at a lower cost.

https://medium.com/@etraining/distributed-computing-processing-models-a5825f12329e

## Parallel Computing & Distributed Computing

Parallel computing and distributed computing are ways of exploiting parallelism in computing to achieve higher performance. Multiple processing elements are used to solve a problem, either to have it done faster or to have a larger size problem been solved. To state simply, if the processing elements share the memory, it is called parallel computing, otherwise it is called distributed computing. Some have opinion that distributed computing is a special form of parallel computing.

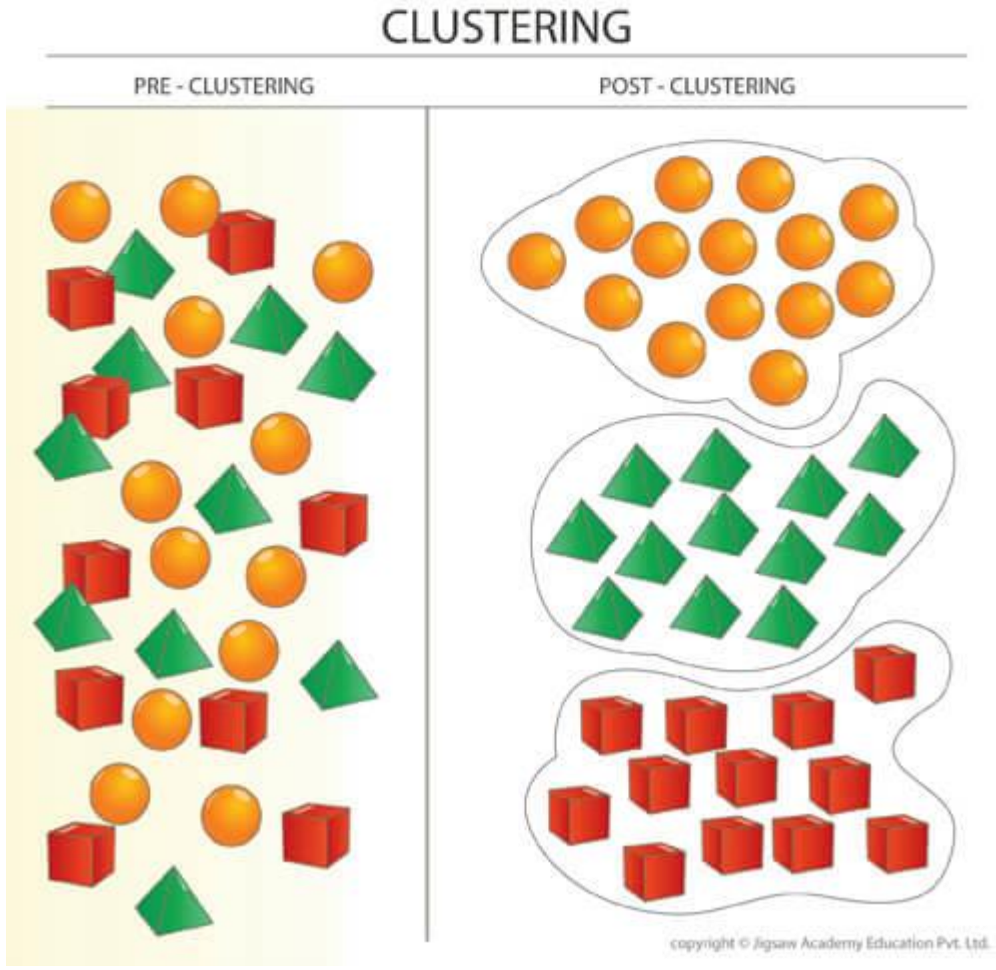


https://subscription.packtpub.com/book/application_development/9781787126992/1/ch01lvl1sec10/parallel-versus-distributed-computing

## K-means Clustering

## Clustering

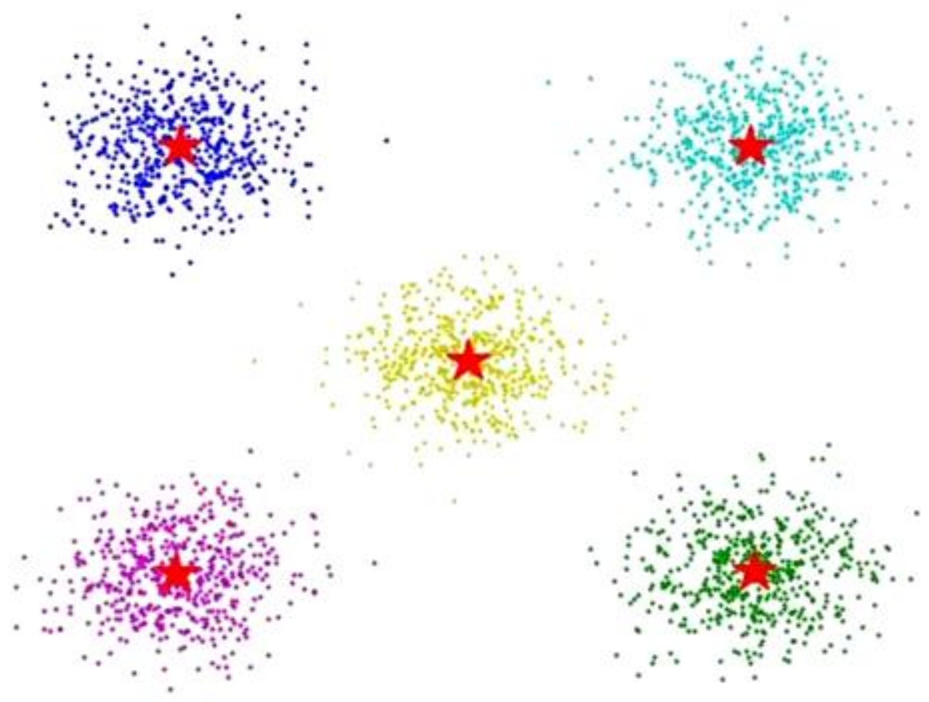
- **Clustering** is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters). ----Wiki



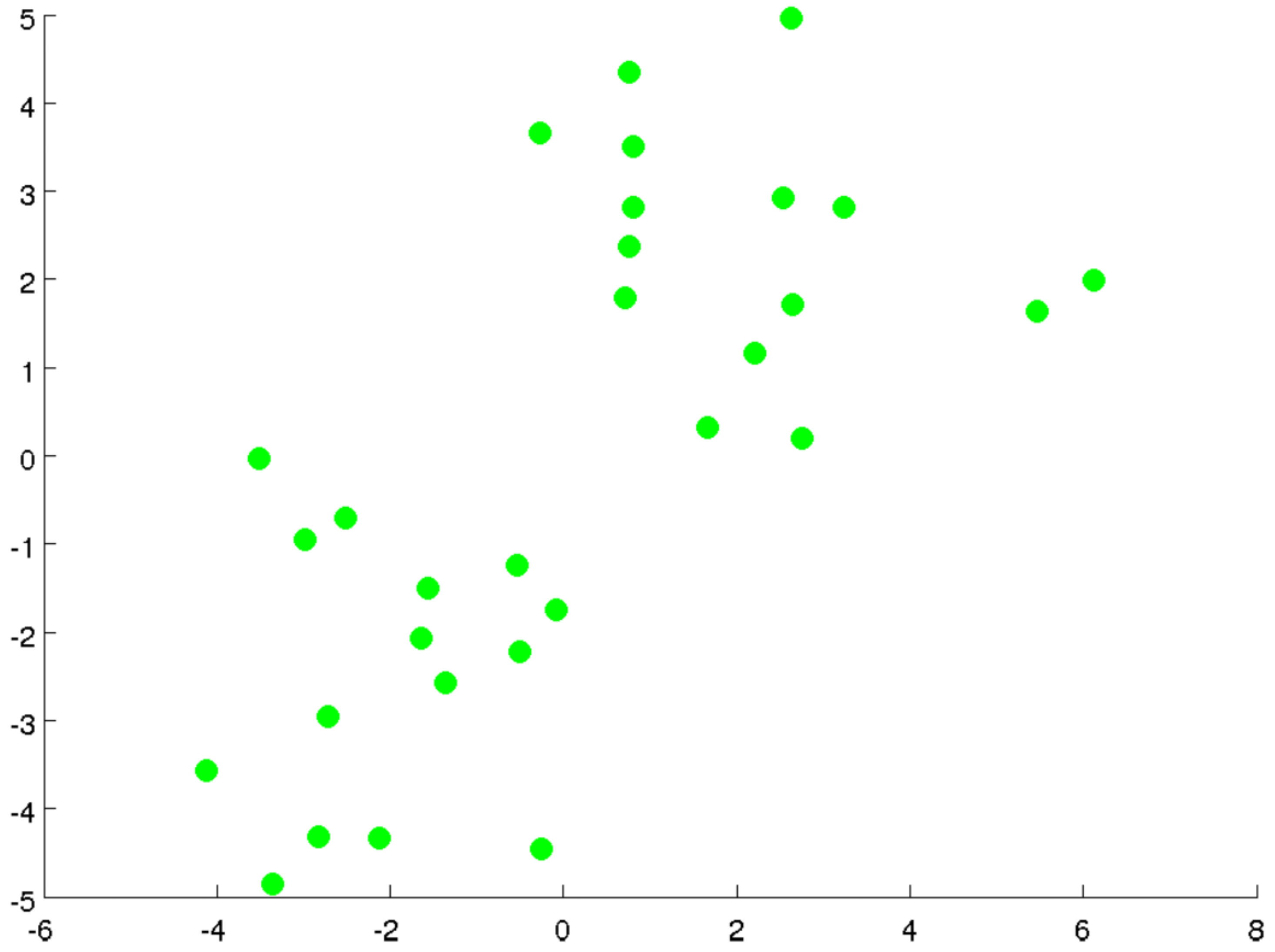
https://analyticstraining.com/cluster-analysis-for-business/

# K-means Clustering

The K-means 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. This algorithm requires the number of clusters to be specified. It scales well to large number of samples and has been used across a large range of application areas in many different fields.



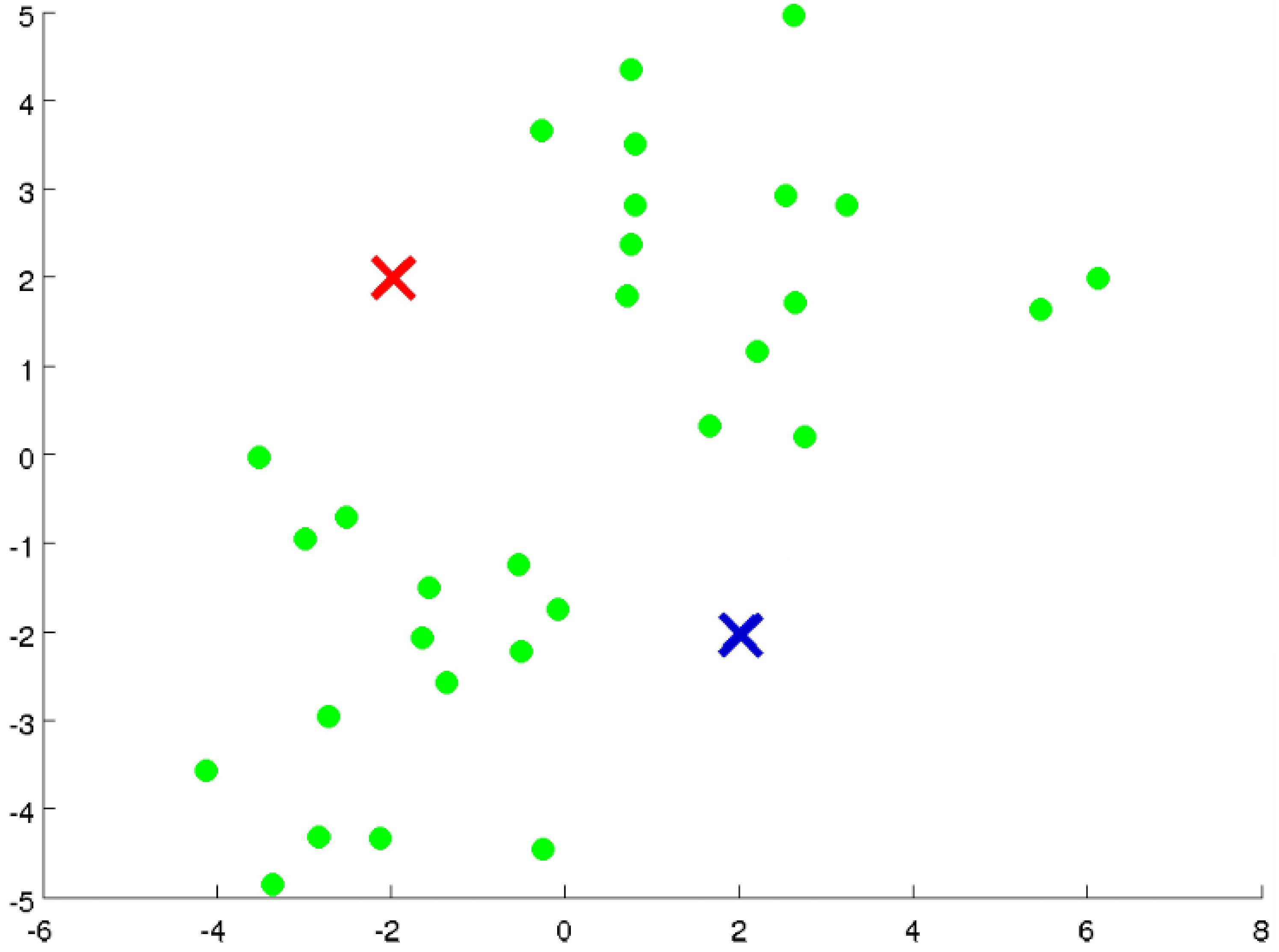
# K-means Clustering



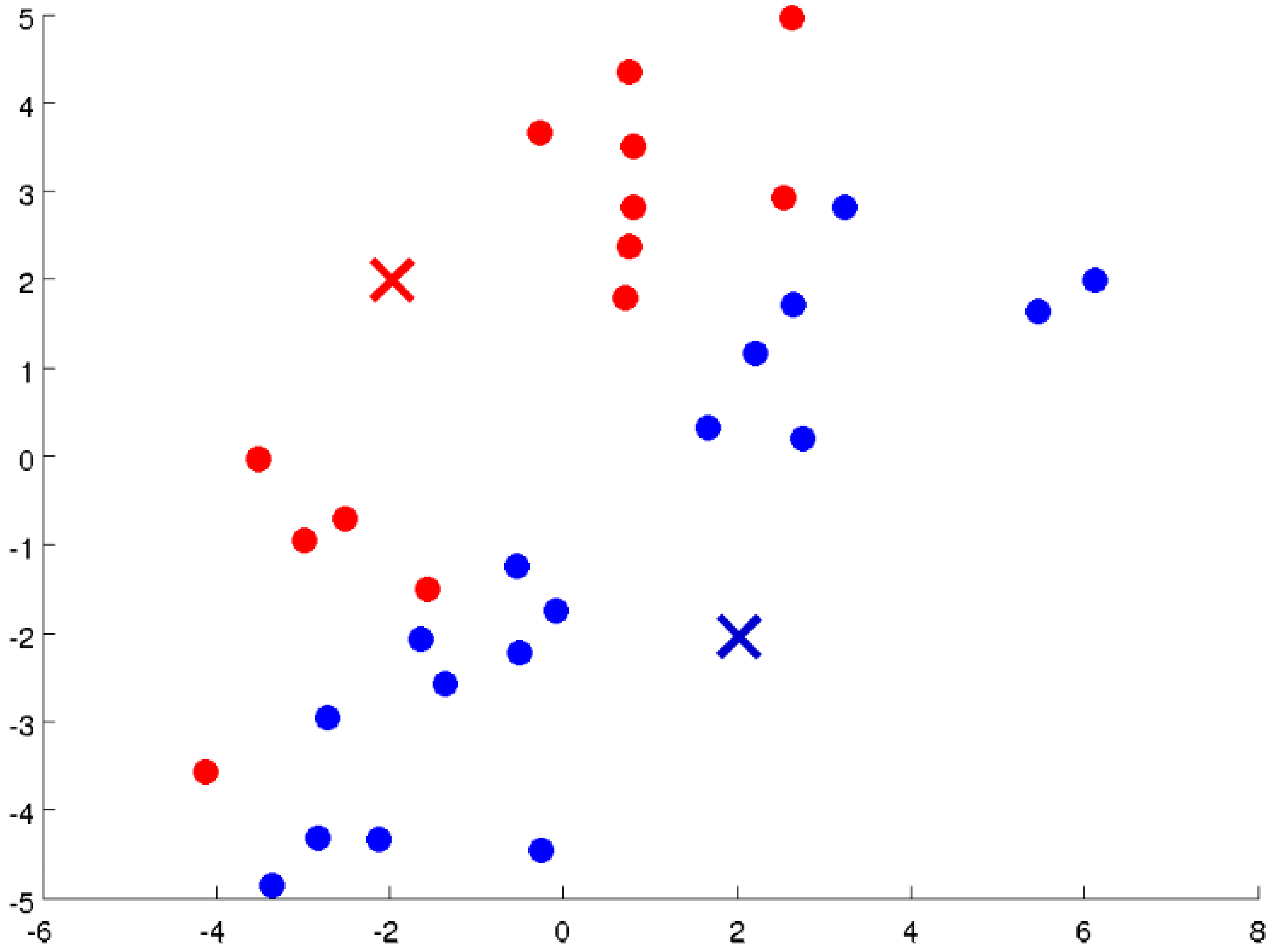
## K-means Clustering

- Step1: Choose K initial centers for clusters
- Step2: In *i*-th iteration, for each point, assign this point to the cluster that has shortest distance between this point and the center.
- Step3: Update the new cluster center by taking the average of the points belonging to this cluster.
- Step4: If the maximum number of iteration reaches, exit the algorithm; or compute the error to decide whether to repeat step2&3

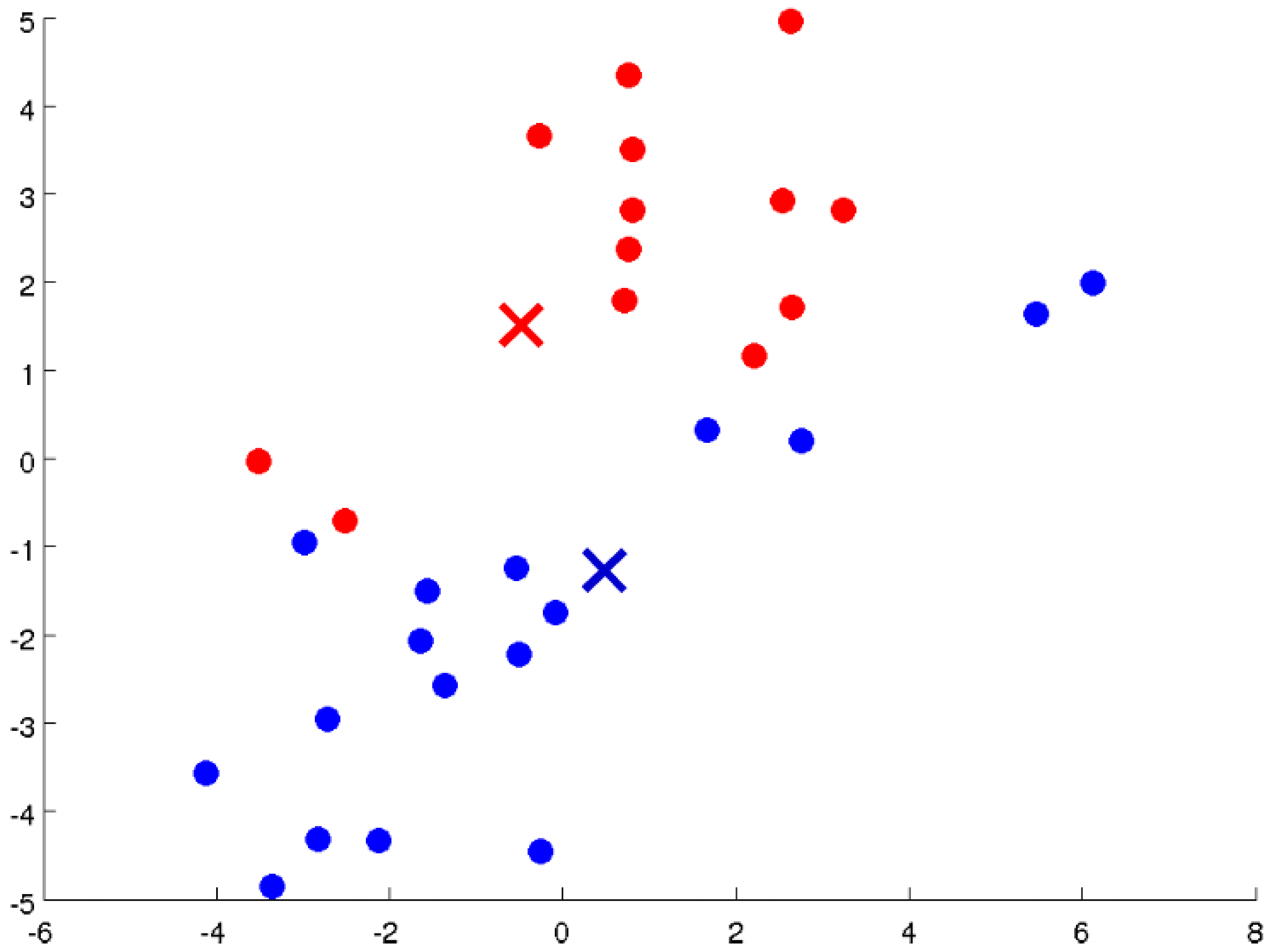
# K-means Clustering



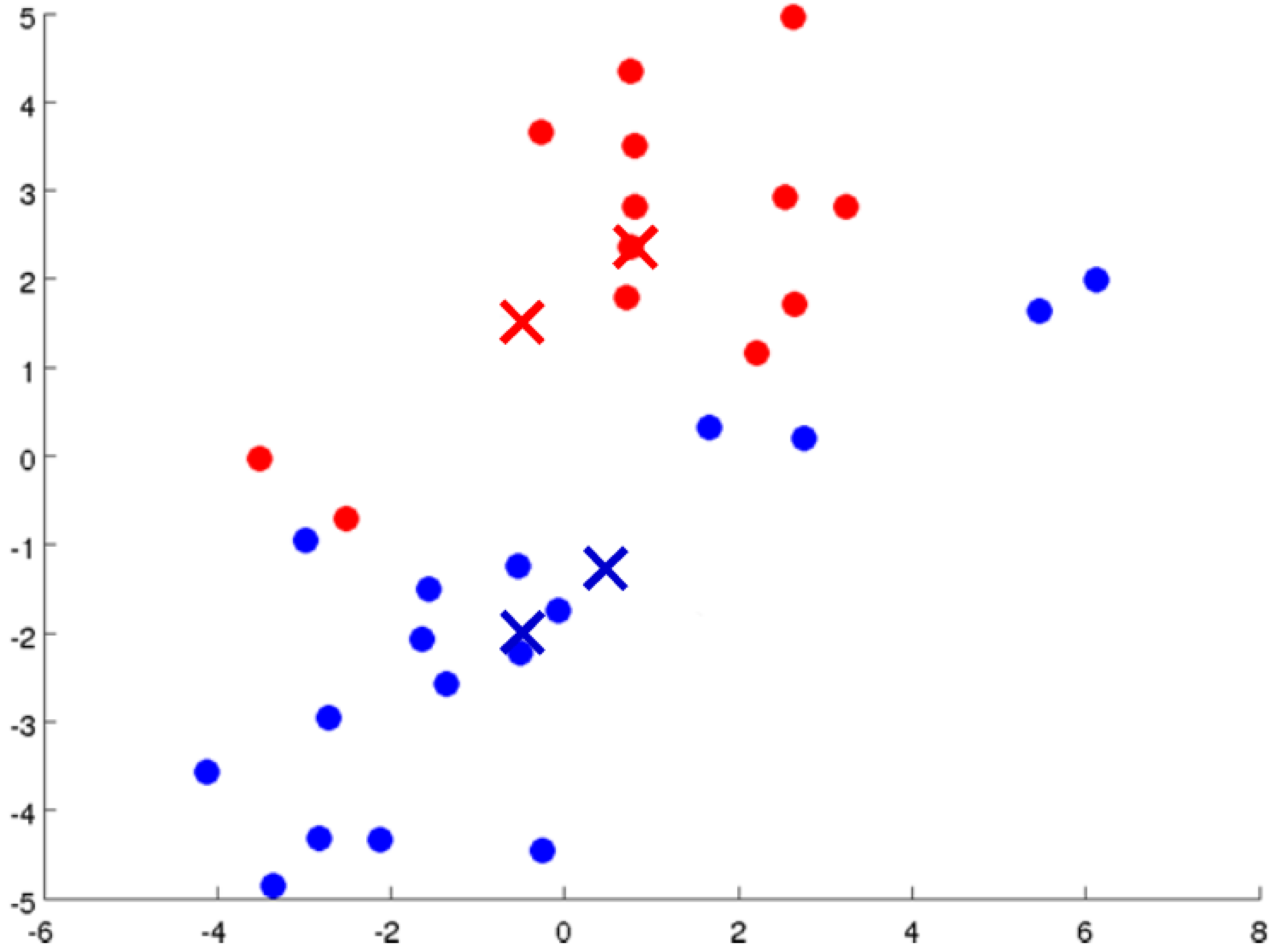
# K-means Clustering



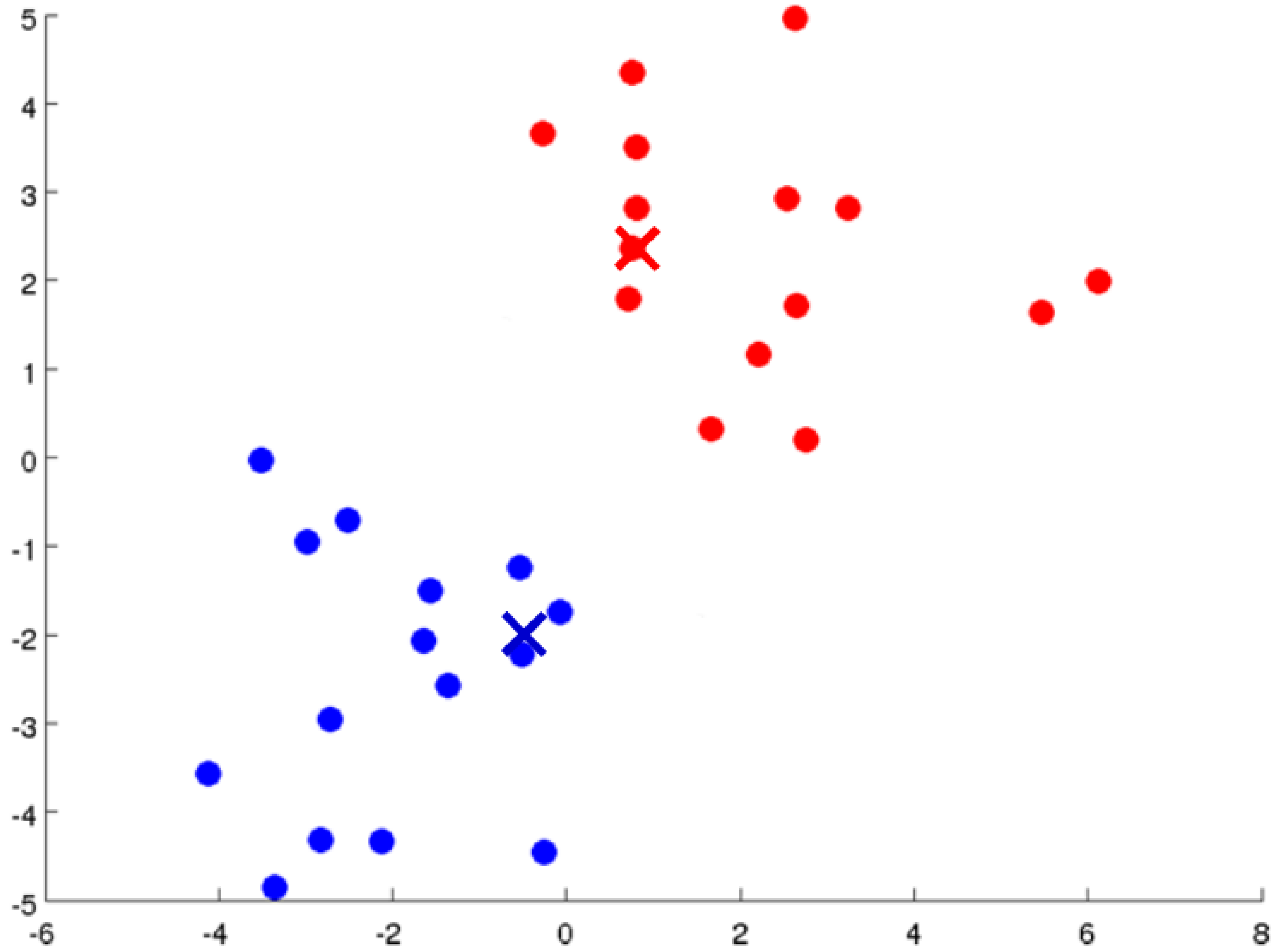
# K-means Clustering



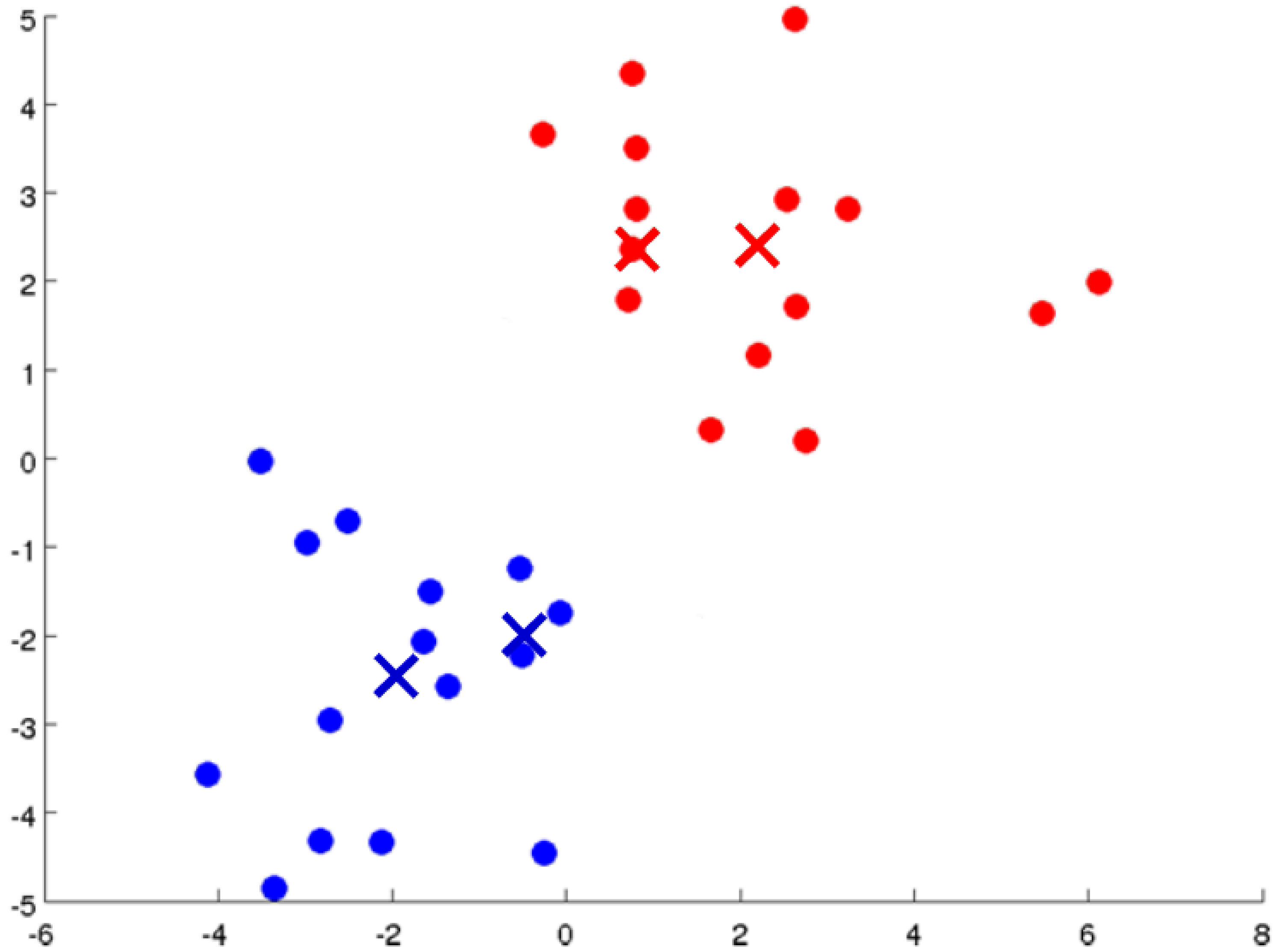
# K-means Clustering



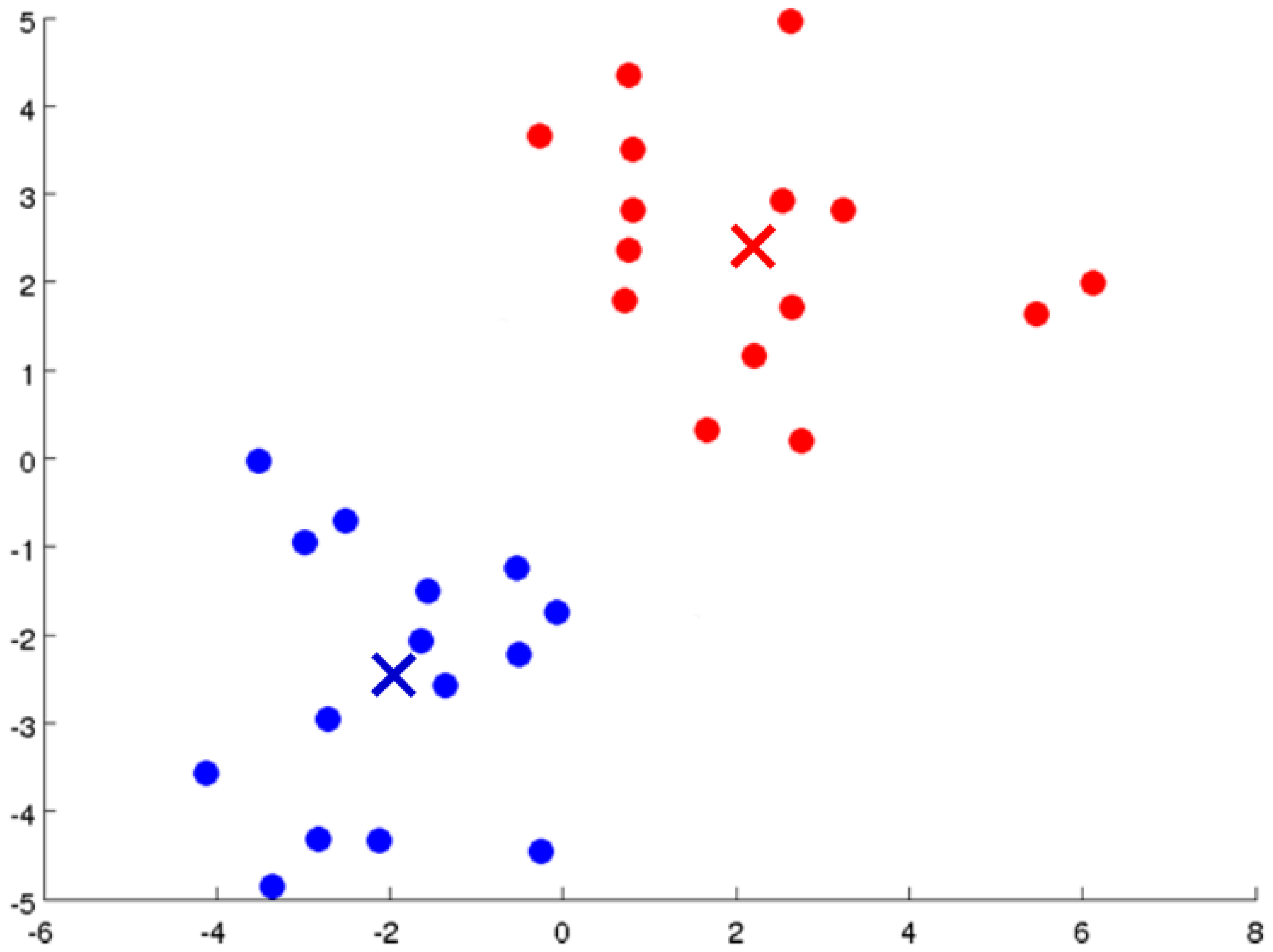
# K-means Clustering



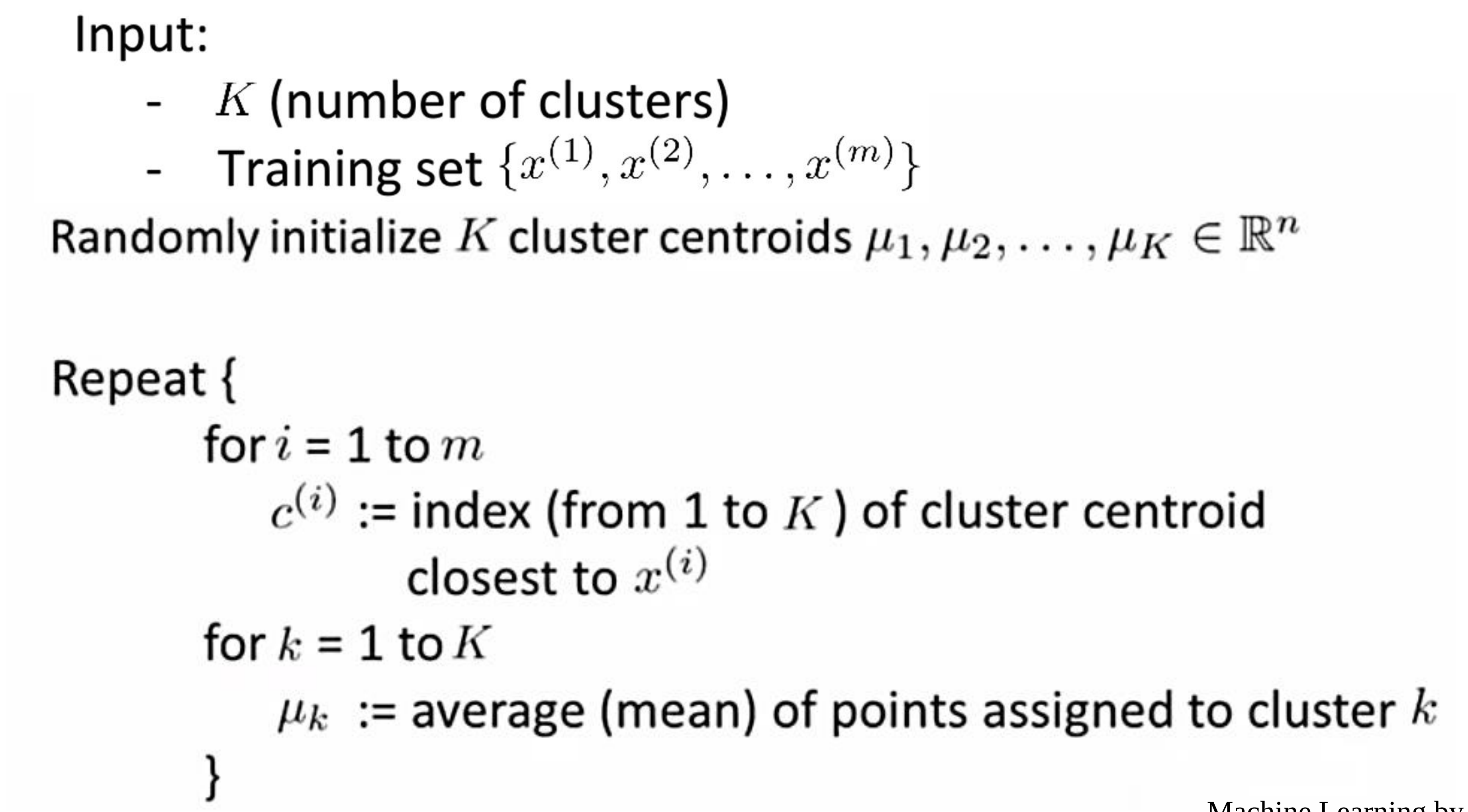
# K-means Clustering



# K-means Clustering



# K-means Clustering



Input:

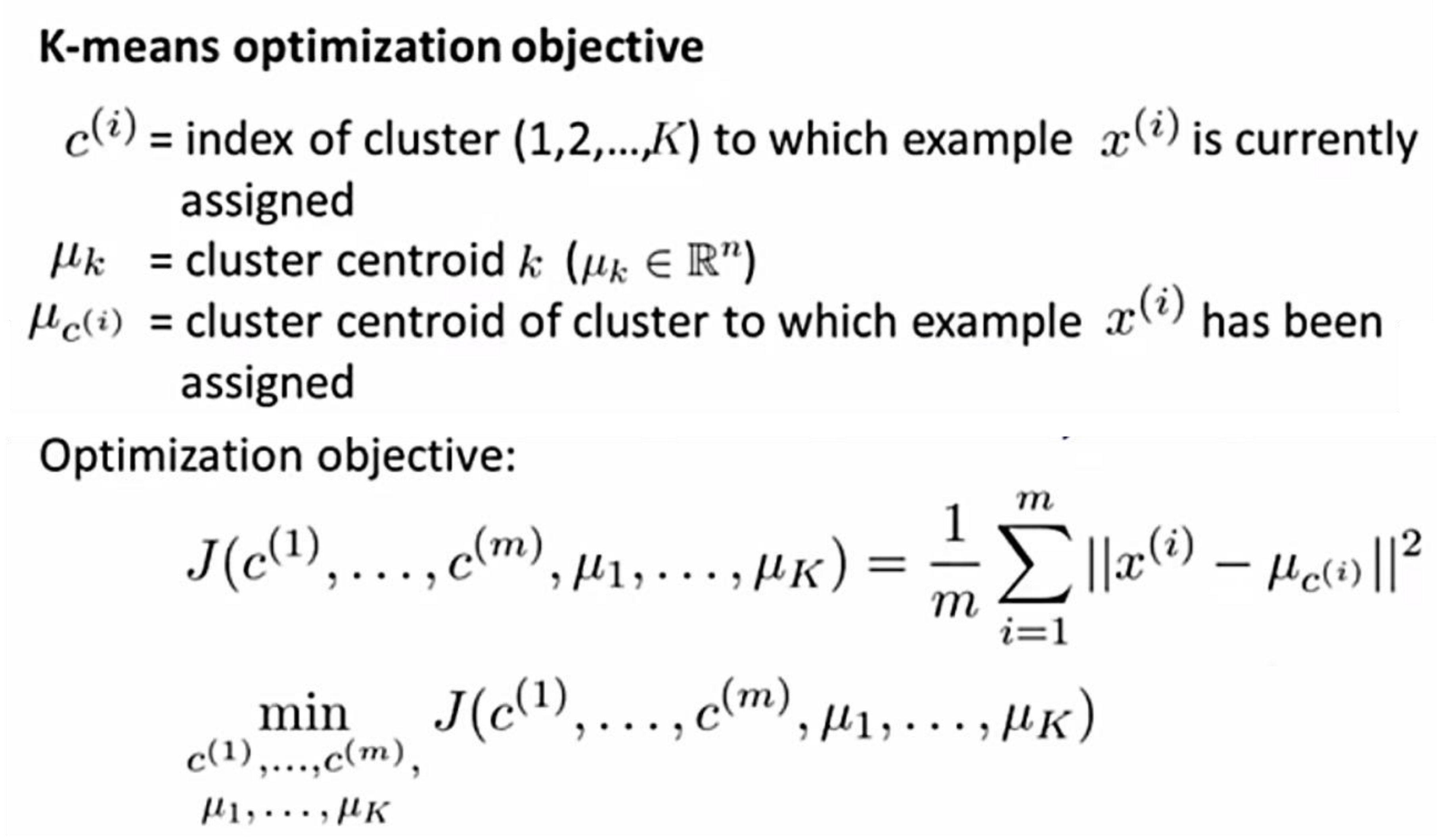
- $K$ (number of clusters)
- Training set $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$

Randomly initialize $K$ cluster centroids $\mu_1, \mu_2, \ldots, \mu_K \in \mathbb{R}^n$

```
Repeat {
        for $i$ = 1 to $m$
            $c^{(i)}$ := index (from 1 to $K$ ) of cluster centroid
                    closest to $x^{(i)}$
        for $k$ = 1 to $K$
            $\mu_k$ := average (mean) of points assigned to cluster $k$
        }
```

Machine Learning by Andrew NG

# K-means Clustering



**K-means optimization objective**

$c^{(i)}$ = index of cluster (1,2,…,$K$) to which example $x^{(i)}$ is currently assigned

$\mu_k$ = cluster centroid $k$ ($\mu_k \in \mathbb{R}^n$)

$\mu_{c^{(i)}}$ = cluster centroid of cluster to which example $x^{(i)}$ has been assigned

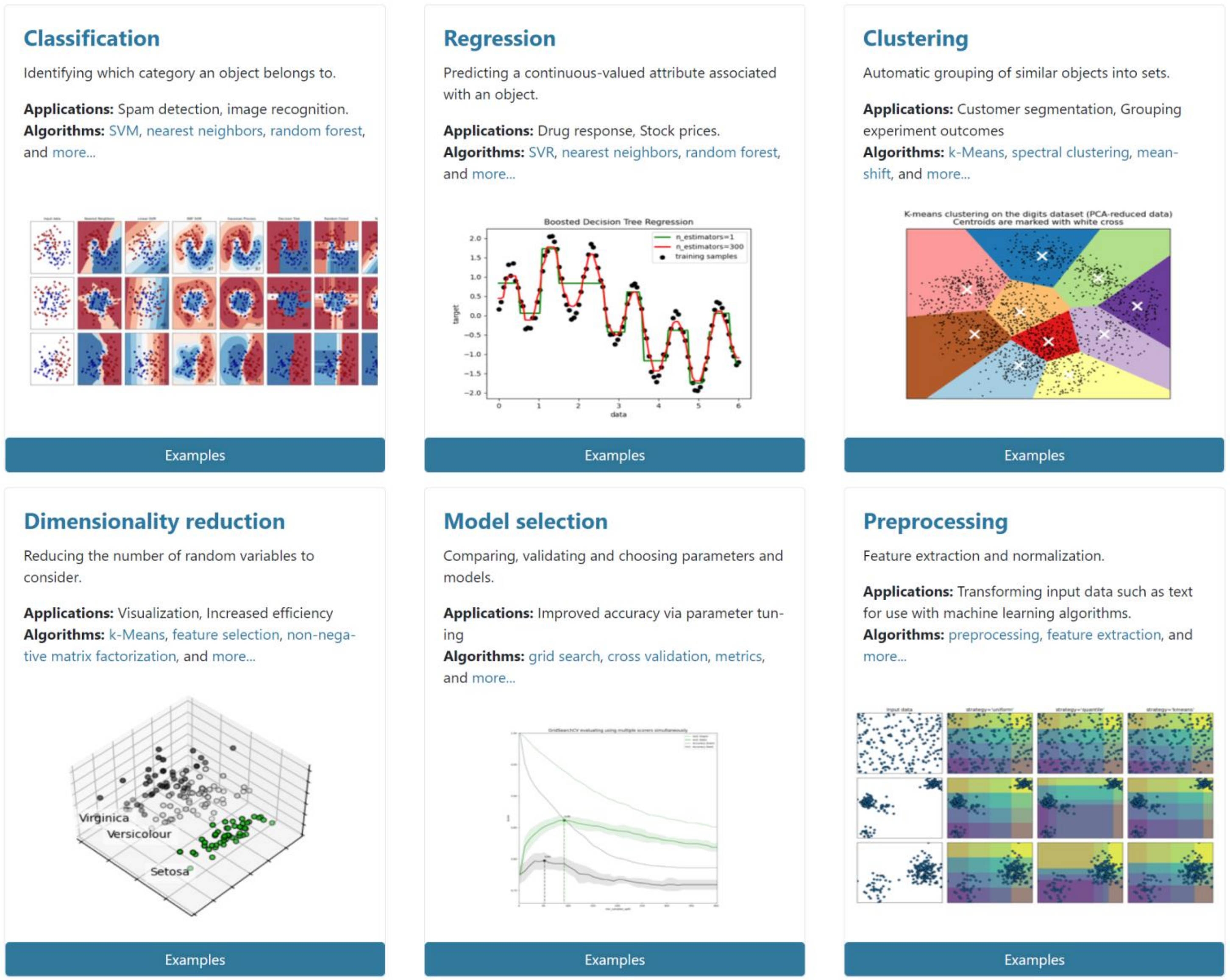
Optimization objective:

$$J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K) = \frac{1}{m} \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2$$

$$\min_{\substack{c^{(1)},\dots,c^{(m)},\\ \mu_1,\dots,\mu_K}} J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$$

## Scikit-learn Package on Python

# Scikit-learn--Machine Learning in Python



### Classification

Identifying which category an object belongs to.

**Applications:** Spam detection, image recognition.
**Algorithms:** SVM, nearest neighbors, random forest, and more...

Examples

### Regression

Predicting a continuous-valued attribute associated with an object.

**Applications:** Drug response, Stock prices.
**Algorithms:** SVR, nearest neighbors, random forest, and more...

Examples

### Clustering

Automatic grouping of similar objects into sets.

**Applications:** Customer segmentation, Grouping experiment outcomes
**Algorithms:** k-Means, spectral clustering, mean-shift, and more...

Examples

### Dimensionality reduction

Reducing the number of random variables to consider.

**Applications:** Visualization, Increased efficiency
**Algorithms:** k-Means, feature selection, non-negative matrix factorization, and more...

Examples

### Model selection

Comparing, validating and choosing parameters and models.

**Applications:** Improved accuracy via parameter tuning
**Algorithms:** grid search, cross validation, metrics, and more...

Examples

### Preprocessing

Feature extraction and normalization.

**Applications:** Transforming input data such as text for use with machine learning algorithms.
**Algorithms:** preprocessing, feature extraction, and more...

Examples

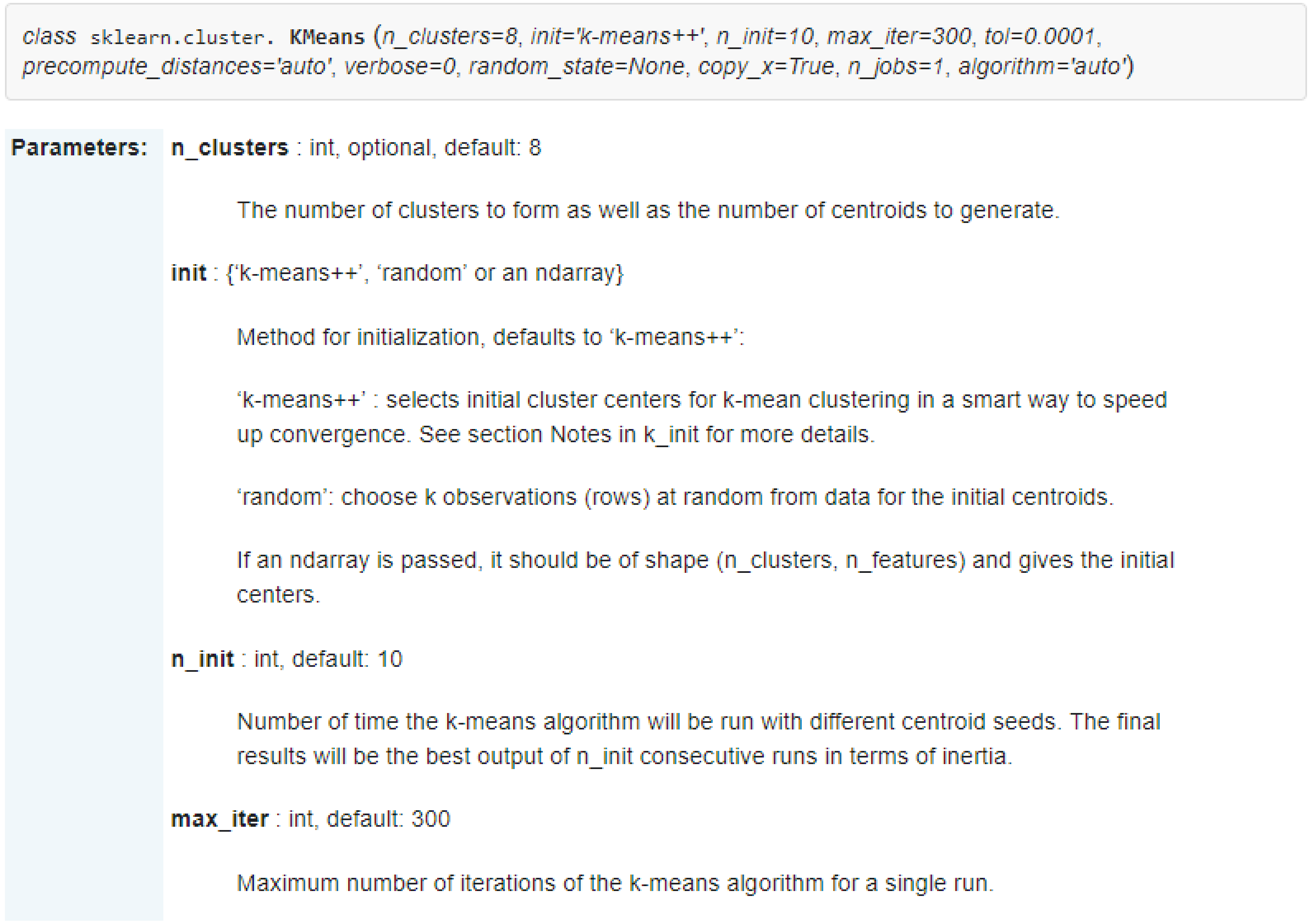
Boosted Decision Tree Regression
n_estimators=1
n_estimators=300
training samples
target
data
K-means clustering on the digits dataset (PCA-reduced data)
Centroids are marked with white cross
Virginica
Versicolour
Setosa

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

https://scikit-learn.org/stable/index.html

https://scikit-learn.org/dev/user_guide.html

# K-means in Scikit-learn



*class* `sklearn.cluster.` **`KMeans`** (*n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001, precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=1, algorithm='auto'*)

**Parameters:**

**n_clusters** : int, optional, default: 8

The number of clusters to form as well as the number of centroids to generate.

**init** : {'k-means++', 'random' or an ndarray}

Method for initialization, defaults to 'k-means++':

'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k_init for more details.

'random': choose k observations (rows) at random from data for the initial centroids.

If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

**n_init** : int, default: 10

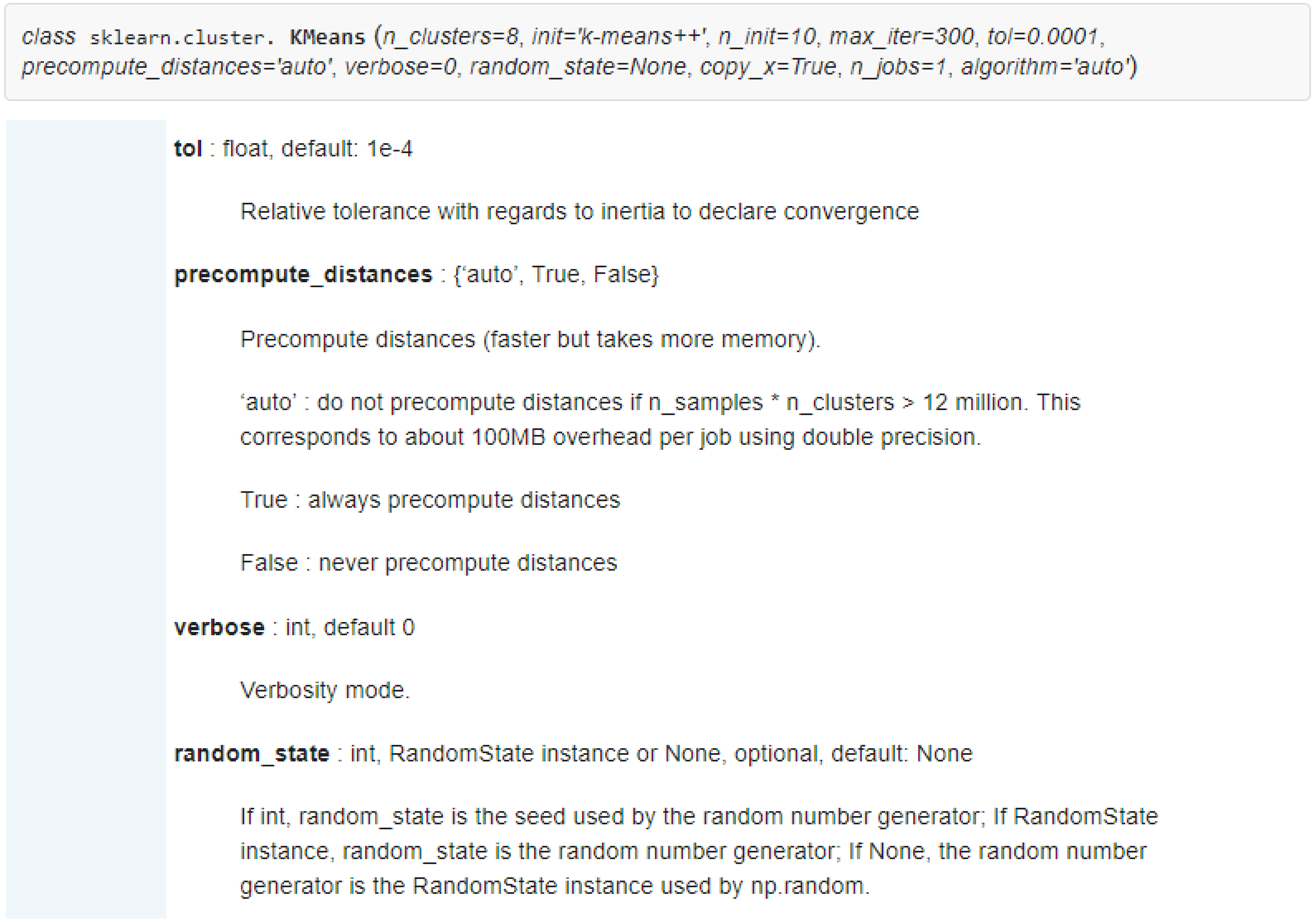
Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

**max_iter** : int, default: 300

Maximum number of iterations of the k-means algorithm for a single run.

https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

# K-means in Scikit-learn



```
class sklearn.cluster. KMeans (n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001,
precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=1, algorithm='auto')
```

**tol** : float, default: 1e-4

Relative tolerance with regards to inertia to declare convergence

**precompute_distances** : {'auto', True, False}

Precompute distances (faster but takes more memory).

'auto' : do not precompute distances if n_samples * n_clusters > 12 million. This corresponds to about 100MB overhead per job using double precision.

True : always precompute distances

False : never precompute distances

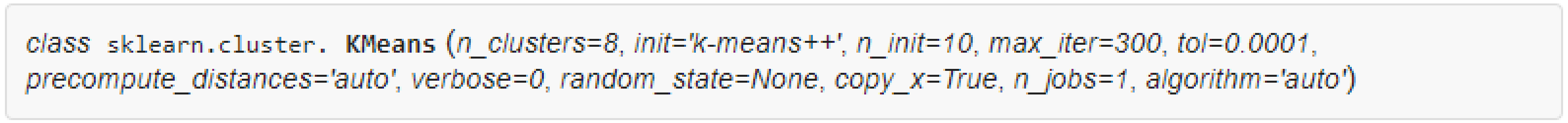
**verbose** : int, default 0

Verbosity mode.

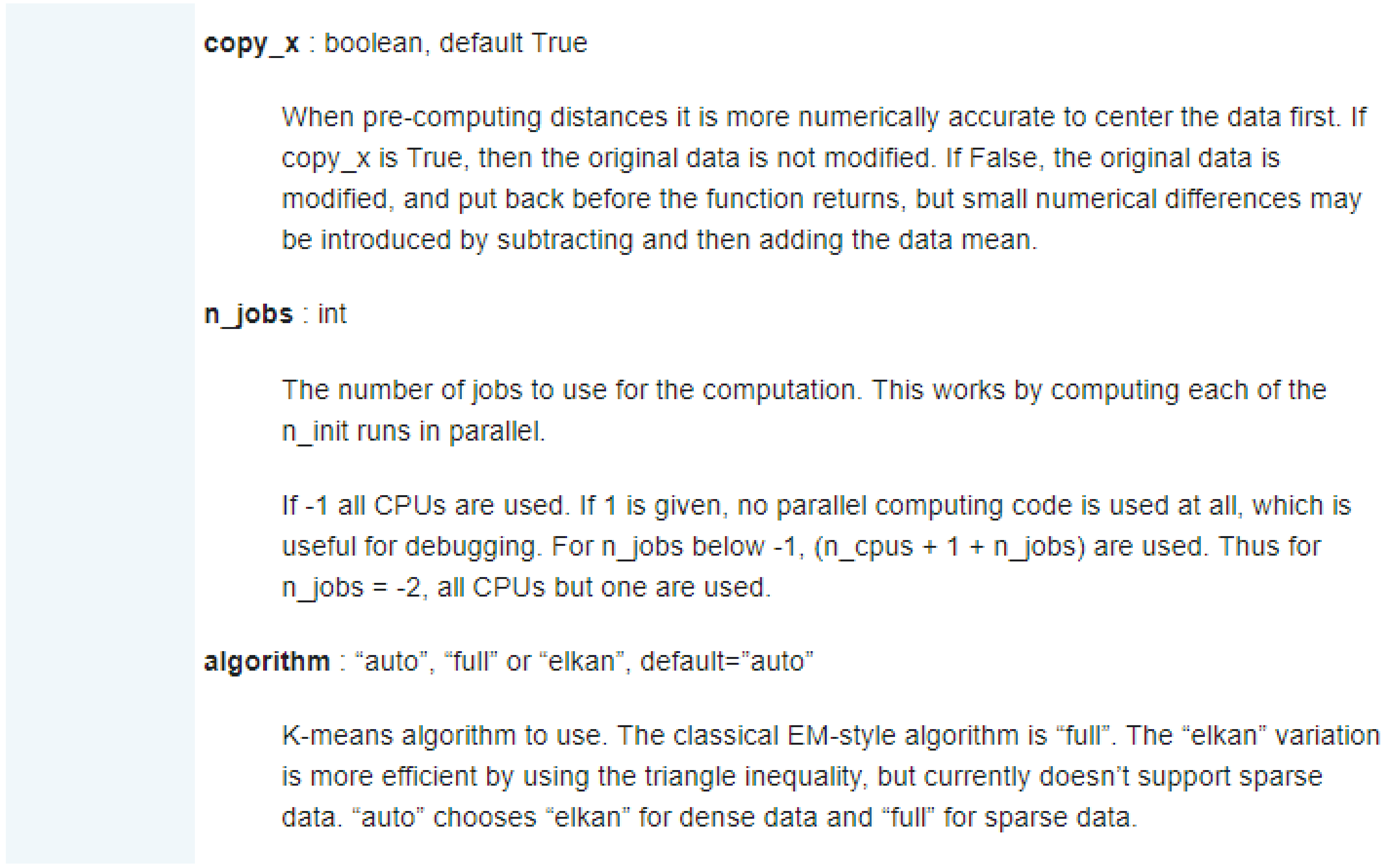
**random_state** : int, RandomState instance or None, optional, default: None

If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

# K-means in Scikit-learn



```
class sklearn.cluster. KMeans (n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001,
precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=1, algorithm='auto')
```



**copy_x** : boolean, default True

When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True, then the original data is not modified. If False, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean.

**n_jobs** : int

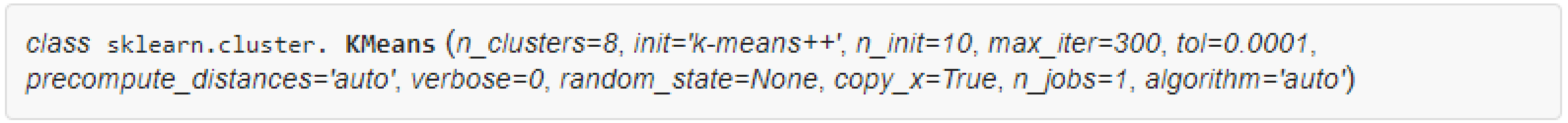
The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

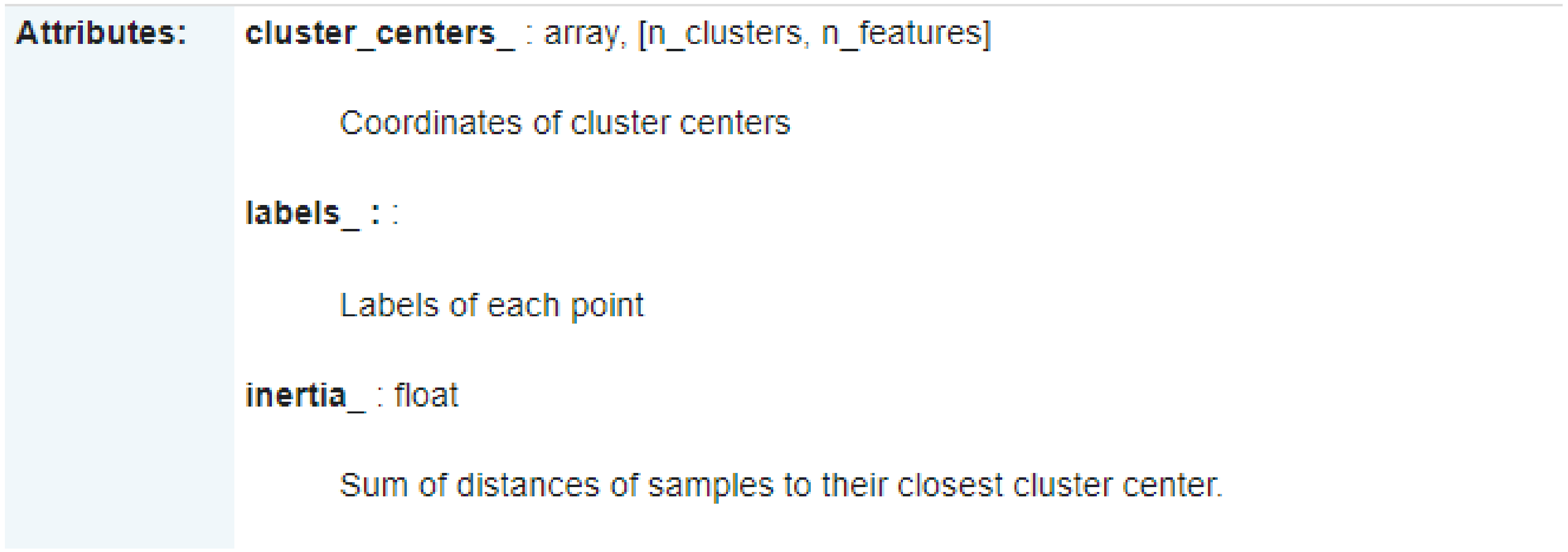
**algorithm** : "auto", "full" or "elkan", default="auto"

K-means algorithm to use. The classical EM-style algorithm is "full". The "elkan" variation is more efficient by using the triangle inequality, but currently doesn't support sparse data. "auto" chooses "elkan" for dense data and "full" for sparse data.

# K-means in Scikit-learn

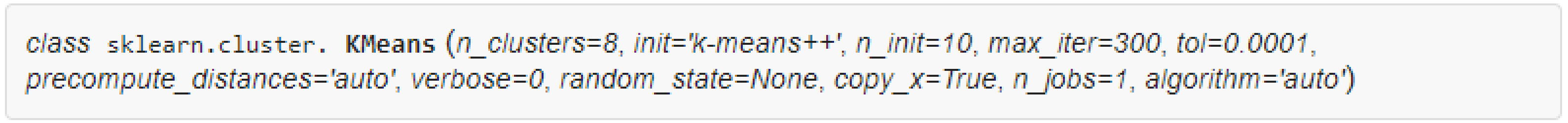


```
class sklearn.cluster. KMeans (n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001,
precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=1, algorithm='auto')
```

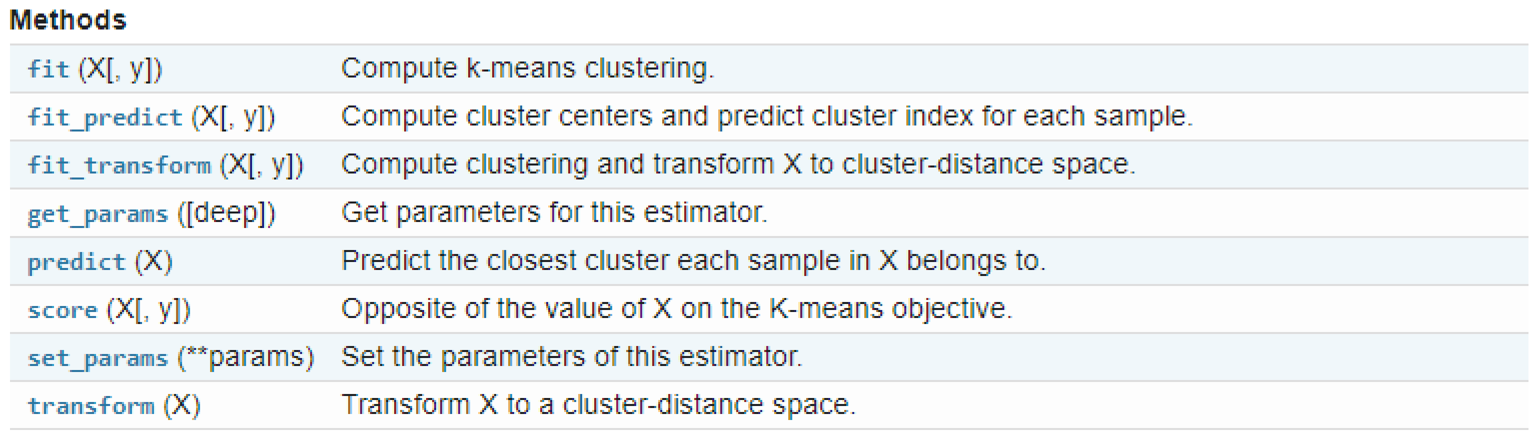


| | |
|---|---|
| **Attributes:** | **cluster_centers_** : array, [n_clusters, n_features]<br><br>Coordinates of cluster centers<br><br>**labels_ :** :<br><br>Labels of each point<br><br>**inertia_** : float<br><br>Sum of distances of samples to their closest cluster center. |

# K-means in Scikit-learn

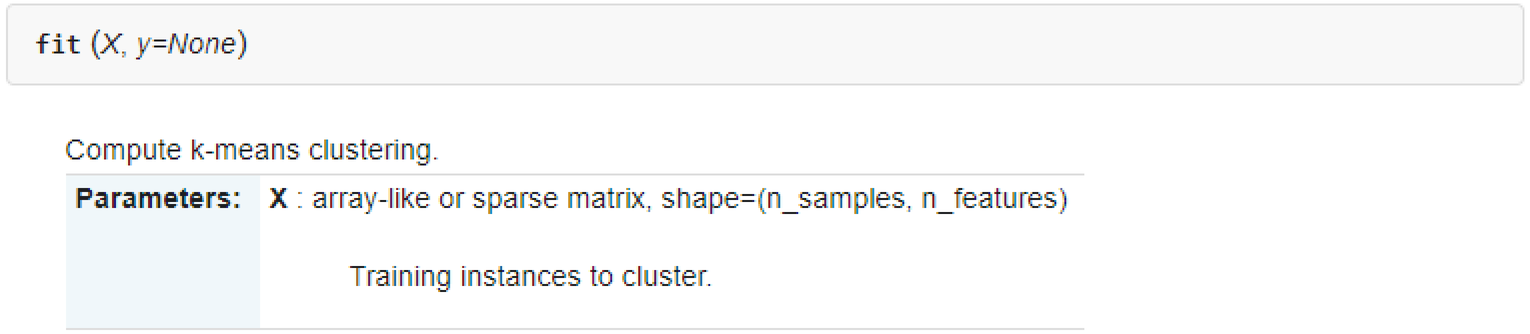


*class* `sklearn.cluster.` **`KMeans`** (*n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001, precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=1, algorithm='auto'*)



**Methods**

| | |
|---|---|
| `fit` (X[, y]) | Compute k-means clustering. |
| `fit_predict` (X[, y]) | Compute cluster centers and predict cluster index for each sample. |
| `fit_transform` (X[, y]) | Compute clustering and transform X to cluster-distance space. |
| `get_params` ([deep]) | Get parameters for this estimator. |
| `predict` (X) | Predict the closest cluster each sample in X belongs to. |
| `score` (X[, y]) | Opposite of the value of X on the K-means objective. |
| `set_params` (**params) | Set the parameters of this estimator. |
| `transform` (X) | Transform X to a cluster-distance space. |

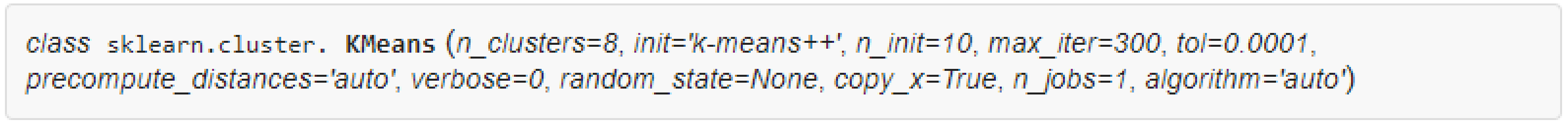


**`fit`** (*X, y=None*)

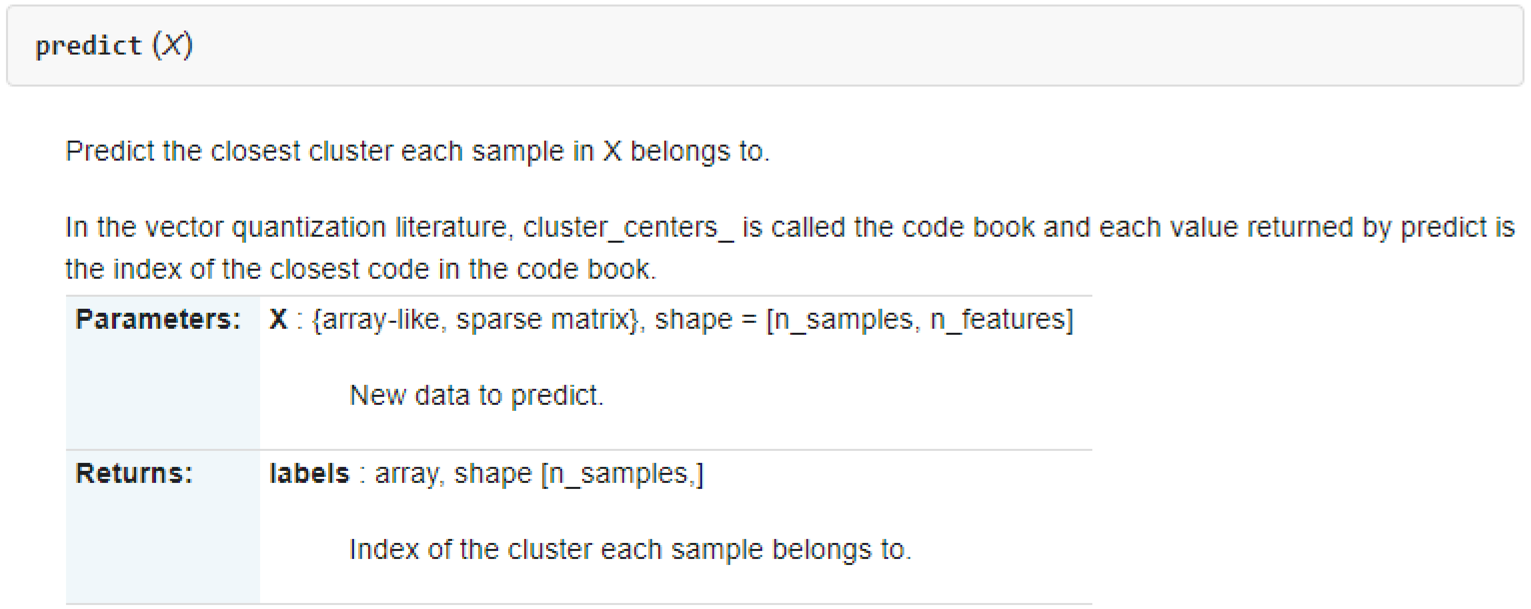
Compute k-means clustering.

| | |
|---|---|
| **Parameters:** | **X** : array-like or sparse matrix, shape=(n_samples, n_features)<br><br>Training instances to cluster. |

# K-means in Scikit-learn



*class* `sklearn.cluster.` **`KMeans`** (*n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001, precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=1, algorithm='auto'*)



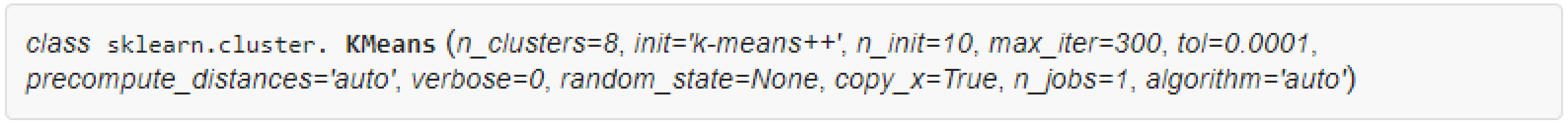
**`predict`** (*X*)

Predict the closest cluster each sample in X belongs to.

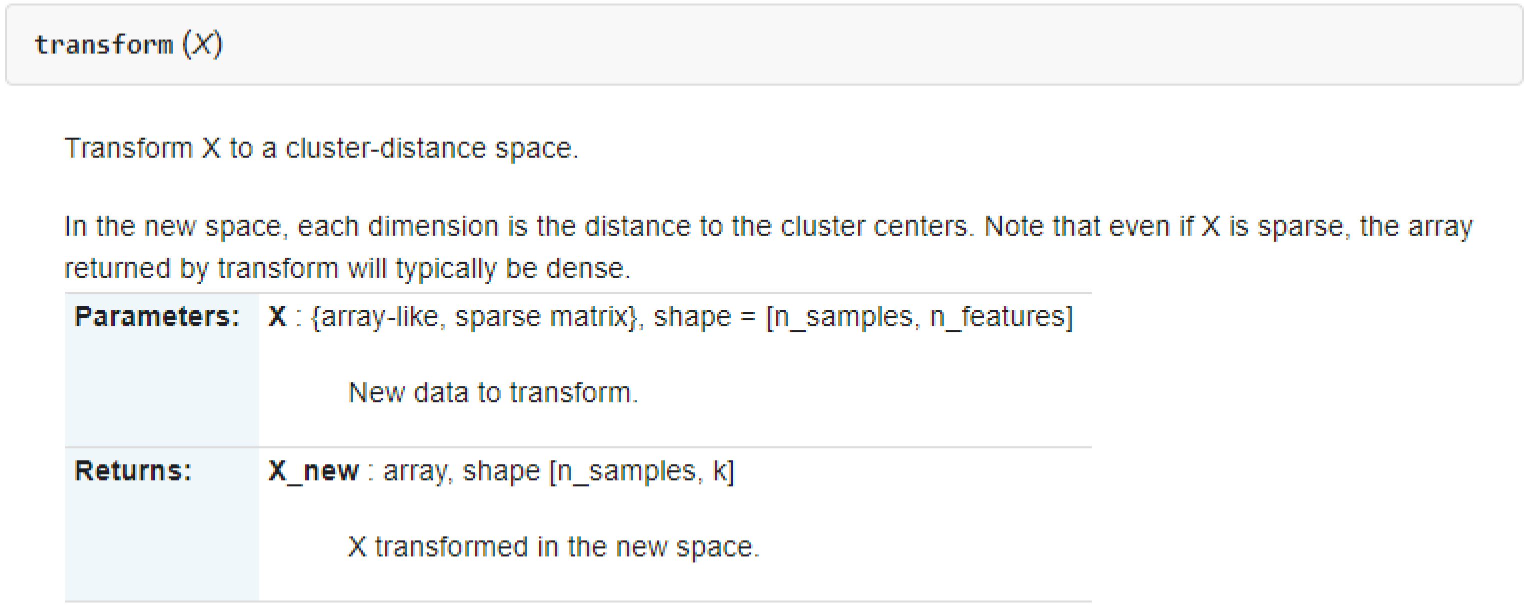
In the vector quantization literature, cluster_centers_ is called the code book and each value returned by predict is the index of the closest code in the code book.

| | |
|---|---|
| **Parameters:** | **X** : {array-like, sparse matrix}, shape = [n_samples, n_features]<br><br>New data to predict. |
| **Returns:** | **labels** : array, shape [n_samples,]<br><br>Index of the cluster each sample belongs to. |

# K-means in Scikit-learn



*class* `sklearn.cluster.` **`KMeans`** (*n_clusters=8*, *init='k-means++'*, *n_init=10*, *max_iter=300*, *tol=0.0001*, *precompute_distances='auto'*, *verbose=0*, *random_state=None*, *copy_x=True*, *n_jobs=1*, *algorithm='auto'*)



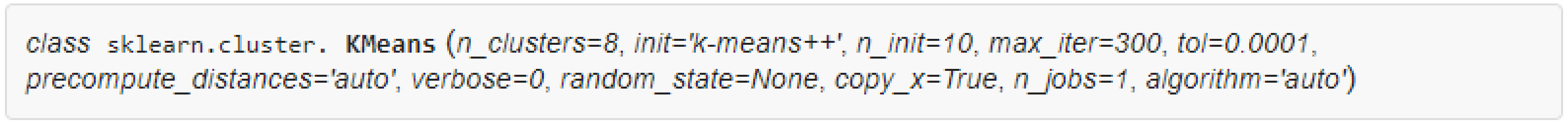
**`transform`** (*X*)

Transform X to a cluster-distance space.

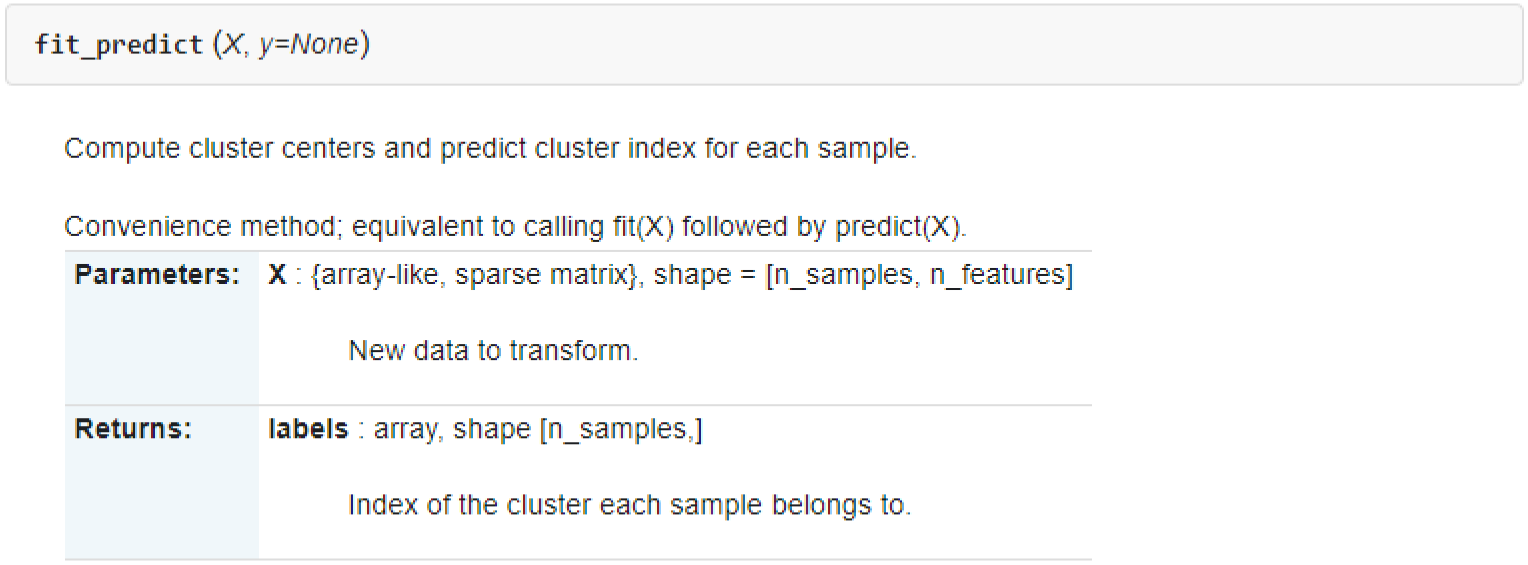
In the new space, each dimension is the distance to the cluster centers. Note that even if X is sparse, the array returned by transform will typically be dense.

| | |
|---|---|
| **Parameters:** | **X** : {array-like, sparse matrix}, shape = [n_samples, n_features]<br><br>New data to transform. |
| **Returns:** | **X_new** : array, shape [n_samples, k]<br><br>X transformed in the new space. |

# K-means in Scikit-learn



```
class sklearn.cluster.KMeans(n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001,
precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=1, algorithm='auto')
```



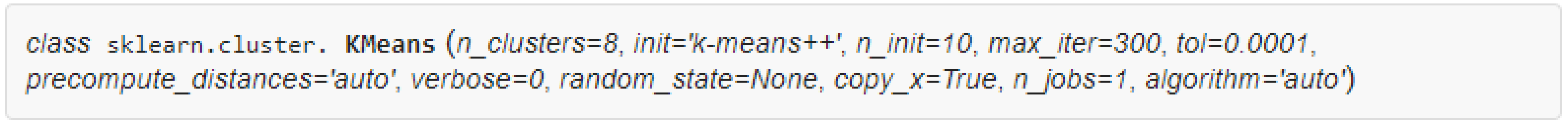
```
fit_predict(X, y=None)
```

Compute cluster centers and predict cluster index for each sample.

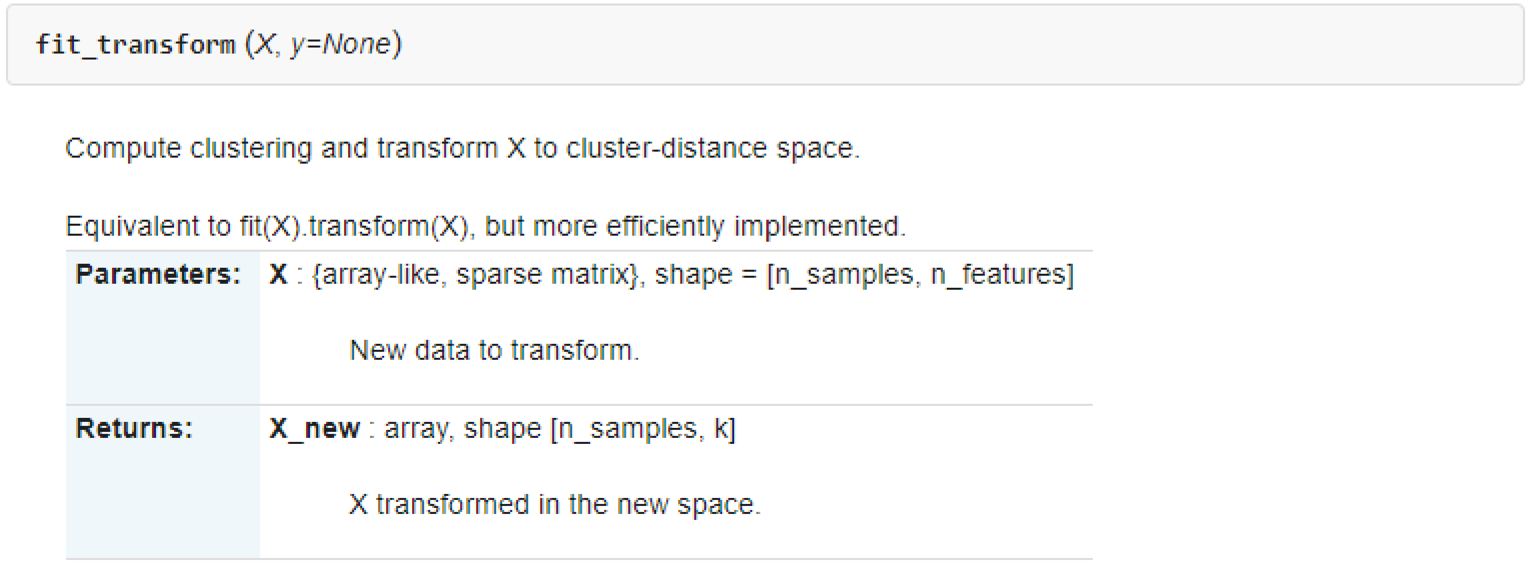
Convenience method; equivalent to calling fit(X) followed by predict(X).

| | |
|---|---|
| **Parameters:** | **X** : {array-like, sparse matrix}, shape = [n_samples, n_features]<br><br>New data to transform. |
| **Returns:** | **labels** : array, shape [n_samples,]<br><br>Index of the cluster each sample belongs to. |

# K-means in Scikit-learn



```
class sklearn.cluster. KMeans (n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001,
precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=1, algorithm='auto')
```



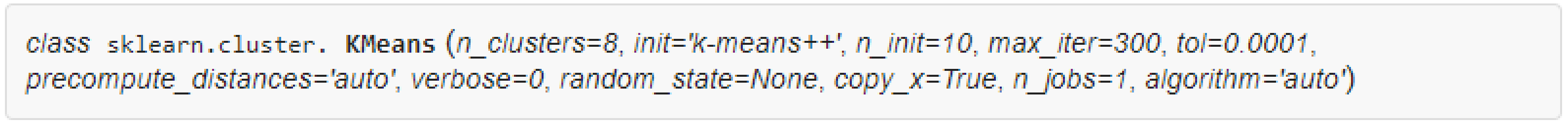
```
fit_transform (X, y=None)
```

Compute clustering and transform X to cluster-distance space.

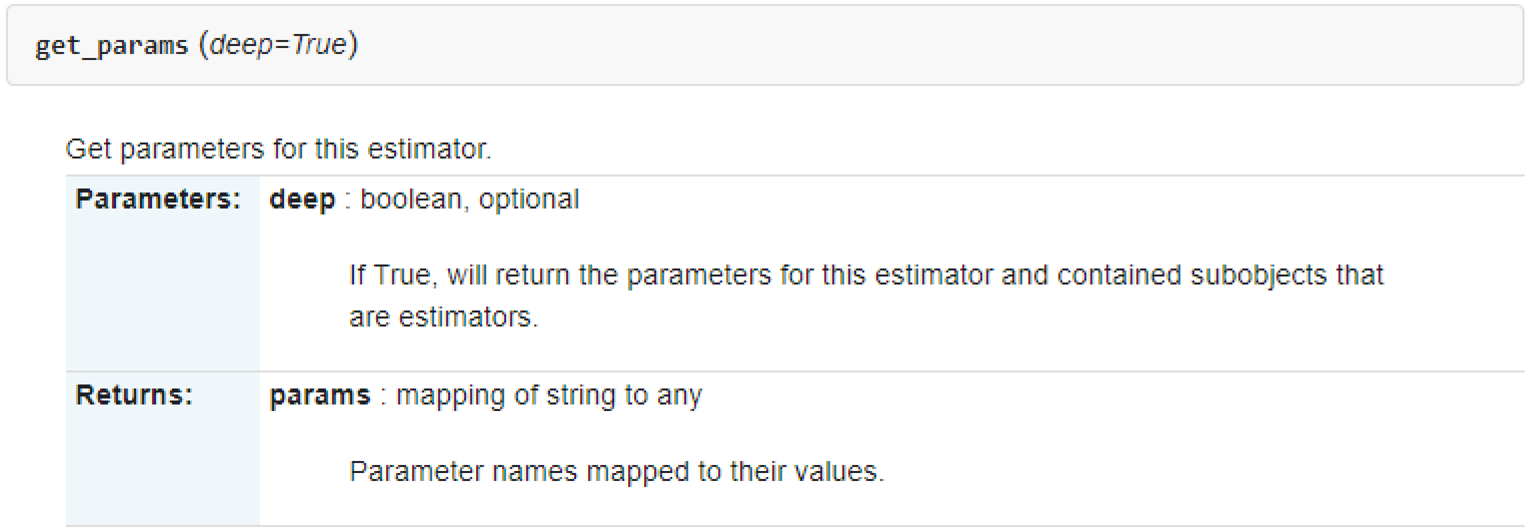
Equivalent to fit(X).transform(X), but more efficiently implemented.

| | |
|---|---|
| **Parameters:** | **X** : {array-like, sparse matrix}, shape = [n_samples, n_features]<br><br>New data to transform. |
| **Returns:** | **X_new** : array, shape [n_samples, k]<br><br>X transformed in the new space. |

# K-means in Scikit-learn



```
class sklearn.cluster. KMeans (n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001,
precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=1, algorithm='auto')
```

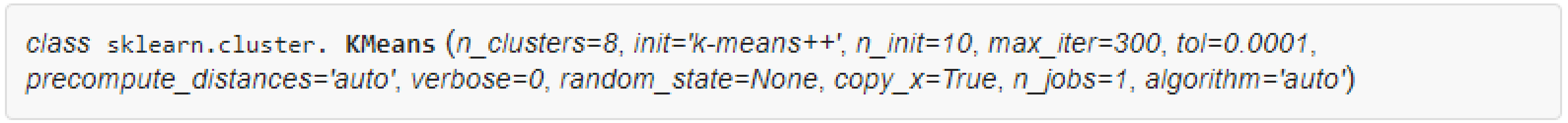


```
get_params (deep=True)
```

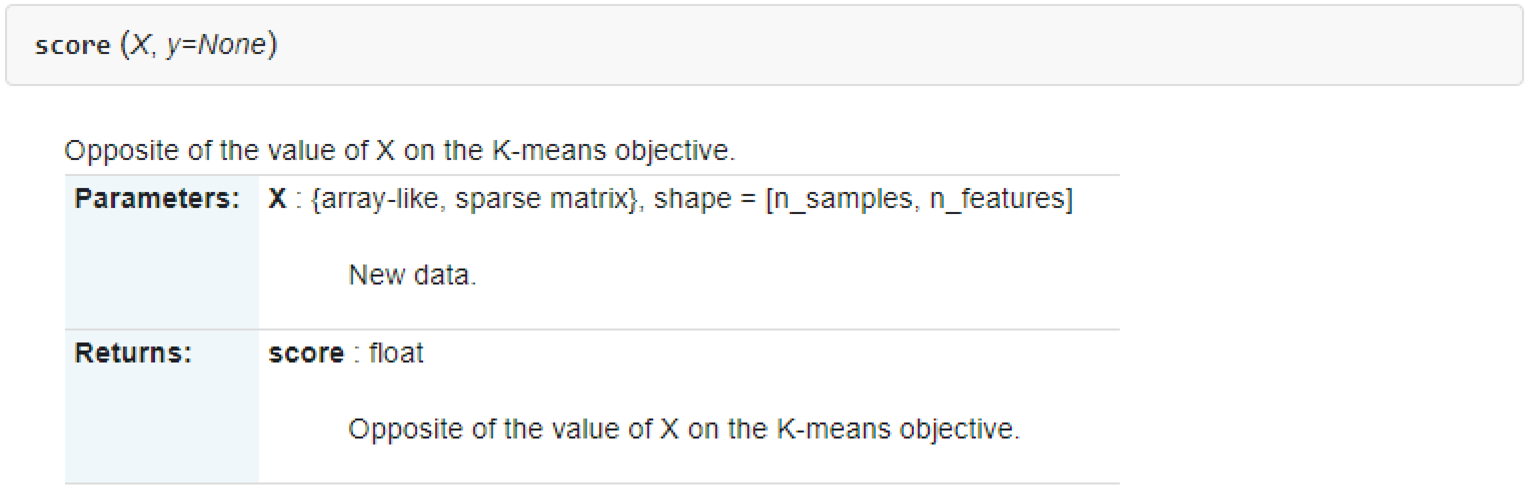
Get parameters for this estimator.

| | |
|---|---|
| **Parameters:** | **deep** : boolean, optional<br><br>If True, will return the parameters for this estimator and contained subobjects that are estimators. |
| **Returns:** | **params** : mapping of string to any<br><br>Parameter names mapped to their values. |

# K-means in Scikit-learn



*class* `sklearn.cluster.` **`KMeans`** (*n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001, precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=1, algorithm='auto'*)

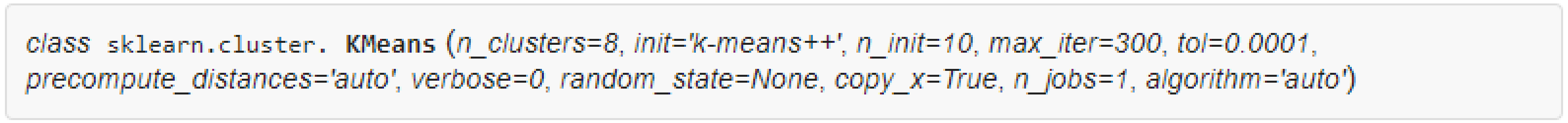


**`score`** (*X, y=None*)

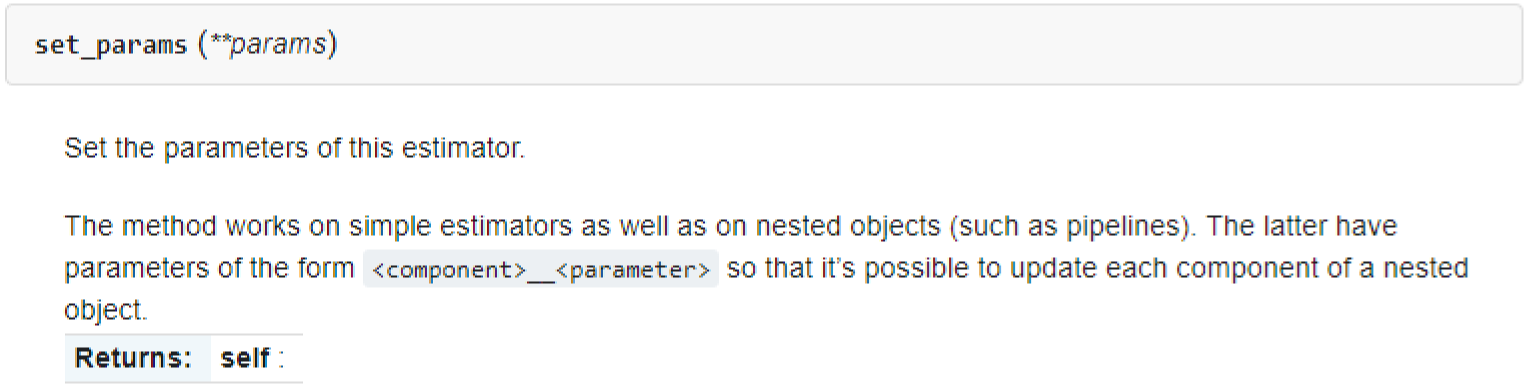
Opposite of the value of X on the K-means objective.

| | |
|---|---|
| **Parameters:** | **X** : {array-like, sparse matrix}, shape = [n_samples, n_features]<br><br>New data. |
| **Returns:** | **score** : float<br><br>Opposite of the value of X on the K-means objective. |

## K-means in Scikit-learn



*class* `sklearn.cluster.` **`KMeans`** (*n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001, precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=1, algorithm='auto'*)



**`set_params`** (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

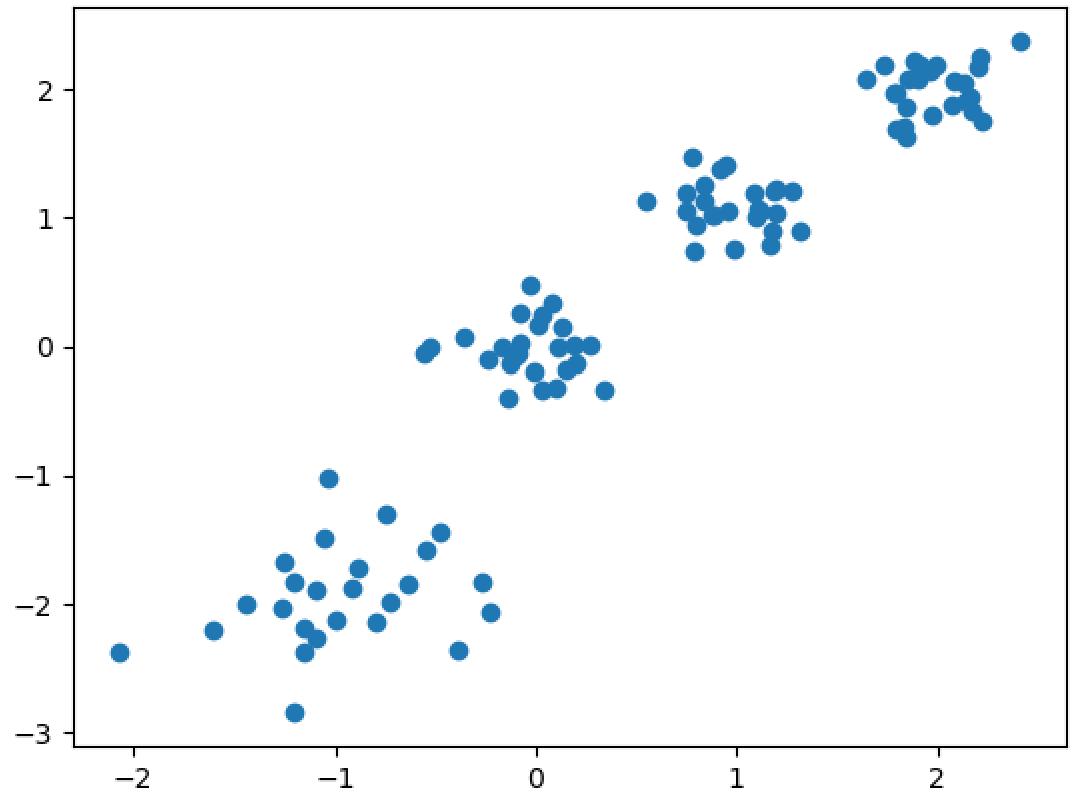
| | |
|---|---|
| **Returns:** | **self** : |

## KMeans Clustering-Example in Python

```
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import KMeans

# Generate dataset by make_blobs
X, y = make_blobs(n_samples=100, n_features=2,centers=[[-1,-2], [0,0], [1,1], [2,2]],cluster_std=[0.4, 0.2, 0.2, 0.2],
random_state =9)

# Define number of cluster
noclusters=4
# Build model and predict
y_pred = KMeans(n_clusters=noclusters).fit_predict(X)

plt.figure()
plt.scatter(X[:, 0], X[:, 1],c=y_pred)
plt.show()
```

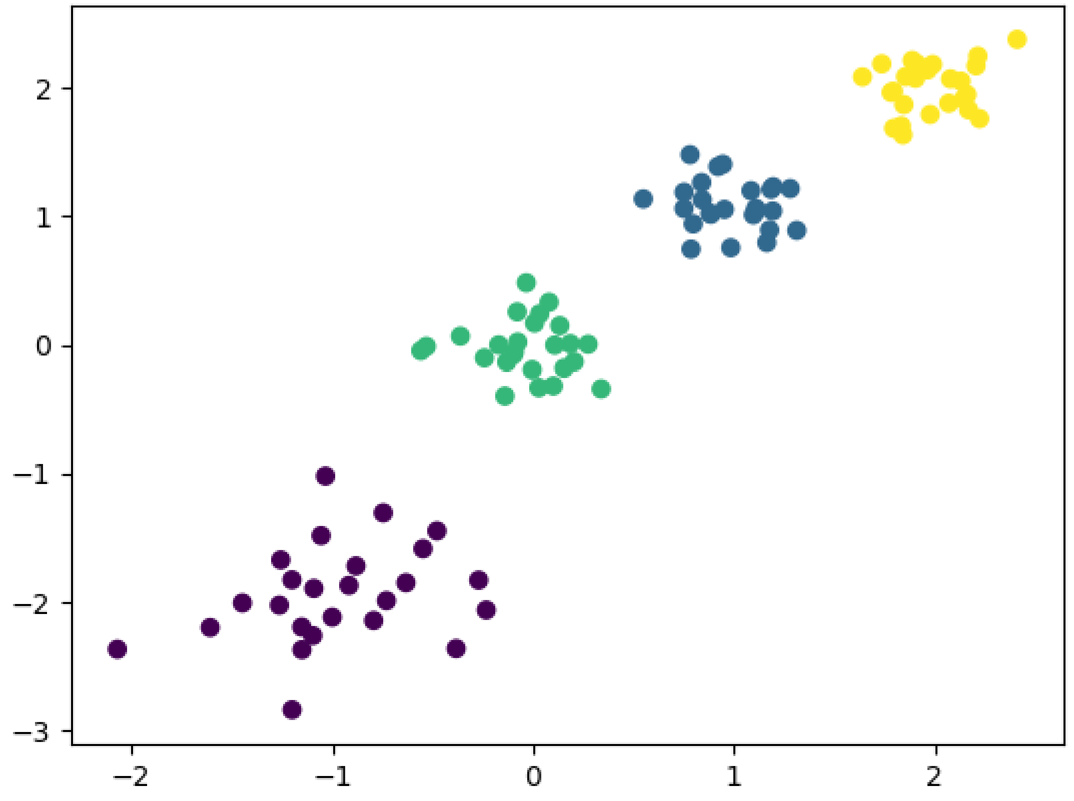
# K-means in Scikit-learn

```
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import make_blobs
# Generate dataset by make_blobs
X, y = make_blobs(n_samples=100, n_features=2,centers=[[-1,-2], [0,0], [1,1], [2,2]],cluster_std=[0.4, 0.2, 0.2, 0.2],
random_state =9)
plt.figure()
plt.scatter(X[:, 0], X[:, 1])
plt.show()
```



# K-means in Scikit-learn

```
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import KMeans
# Generate dataset by make_blobs
X, y = make_blobs(n_samples=100, n_features=2,centers=[[-1,-2], [0,0], [1,1], [2,2]],cluster_std=[0.4, 0.2, 0.2, 0.2],
random_state =9)
# Define number of cluster
noclusters=4
# Build model and predict
y_pred = KMeans(n_clusters=noclusters).fit_predict(X)

plt.figure()
plt.scatter(X[:, 0], X[:, 1],c=y_pred)
plt.show()
```



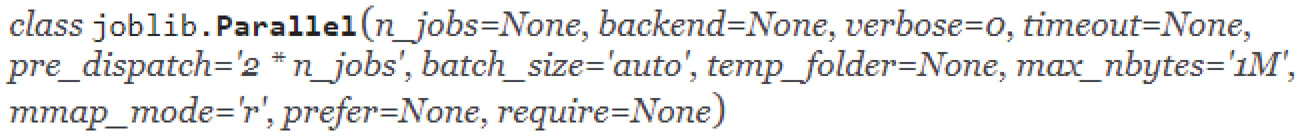
## Joblib Package

## Joblib Package

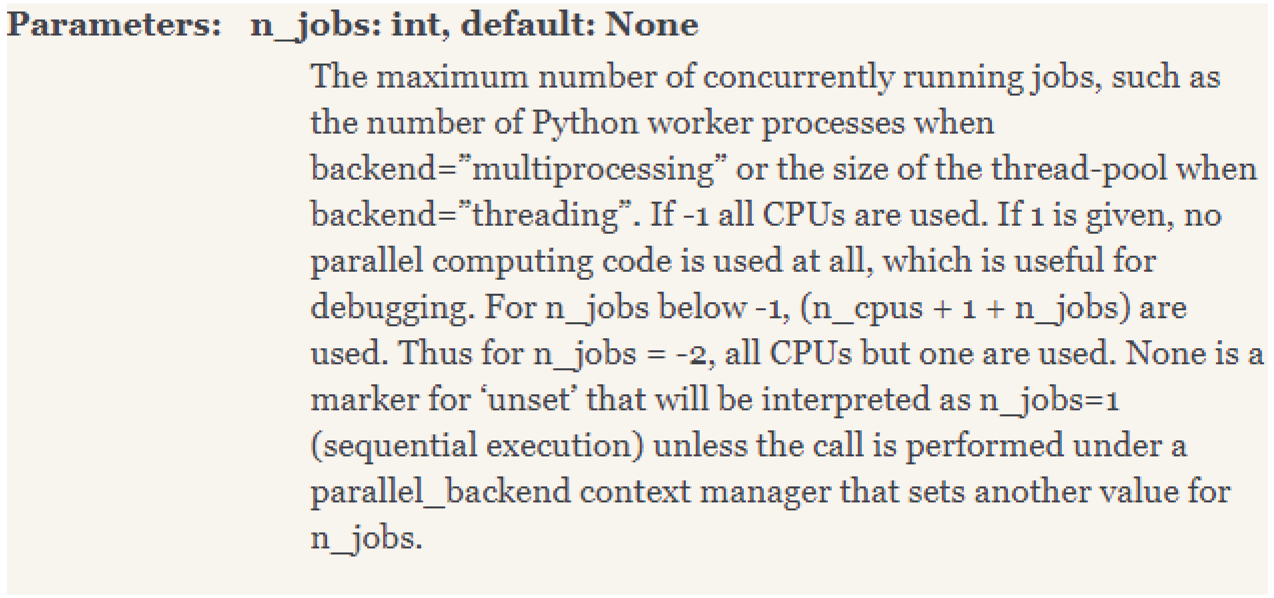
Joblib is a set of tools to provide lightweight pipelining in Python. In particular:

- transparent disk-caching of functions and lazy re-evaluation (memoize pattern)
- easy simple parallel computing

# Joblib Package

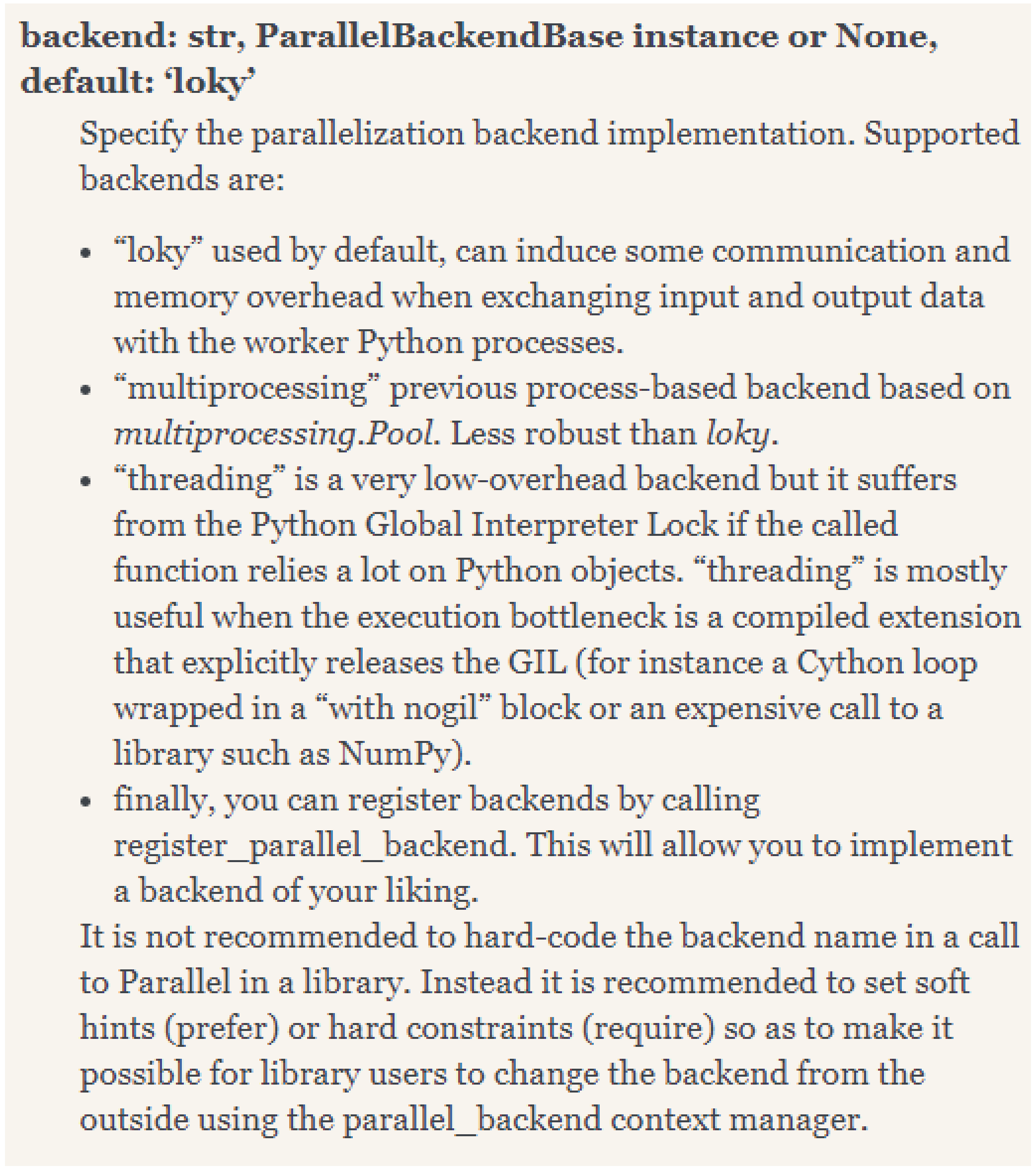


*class* `joblib.`**`Parallel`**(*n_jobs=None, backend=None, verbose=0, timeout=None, pre_dispatch='2 * n_jobs', batch_size='auto', temp_folder=None, max_nbytes='1M', mmap_mode='r', prefer=None, require=None*)



**Parameters:** **n_jobs: int, default: None**

The maximum number of concurrently running jobs, such as the number of Python worker processes when backend="multiprocessing" or the size of the thread-pool when backend="threading". If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used. None is a marker for 'unset' that will be interpreted as n_jobs=1 (sequential execution) unless the call is performed under a parallel_backend context manager that sets another value for n_jobs.



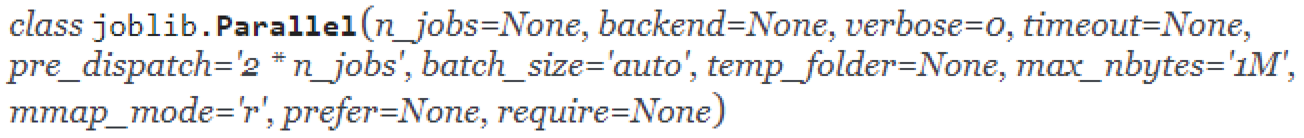
**backend: str, ParallelBackendBase instance or None, default: 'loky'**

Specify the parallelization backend implementation. Supported backends are:

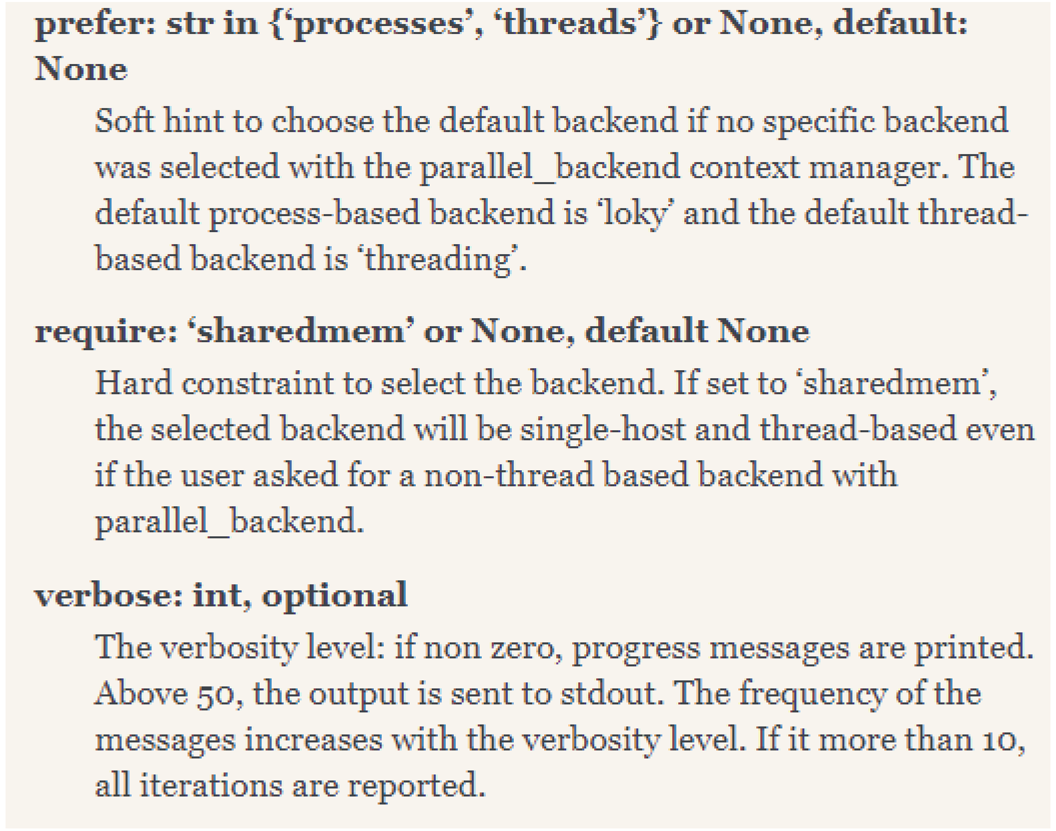
- "loky" used by default, can induce some communication and memory overhead when exchanging input and output data with the worker Python processes.
- "multiprocessing" previous process-based backend based on *multiprocessing.Pool*. Less robust than *loky*.
- "threading" is a very low-overhead backend but it suffers from the Python Global Interpreter Lock if the called function relies a lot on Python objects. "threading" is mostly useful when the execution bottleneck is a compiled extension that explicitly releases the GIL (for instance a Cython loop wrapped in a "with nogil" block or an expensive call to a library such as NumPy).
- finally, you can register backends by calling register_parallel_backend. This will allow you to implement a backend of your liking.

It is not recommended to hard-code the backend name in a call to Parallel in a library. Instead it is recommended to set soft hints (prefer) or hard constraints (require) so as to make it possible for library users to change the backend from the outside using the parallel_backend context manager.

## Joblib Package



*class* `joblib.Parallel`(*n_jobs=None, backend=None, verbose=0, timeout=None, pre_dispatch='2 * n_jobs', batch_size='auto', temp_folder=None, max_nbytes='1M', mmap_mode='r', prefer=None, require=None*)



**prefer: str in {'processes', 'threads'} or None, default: None**

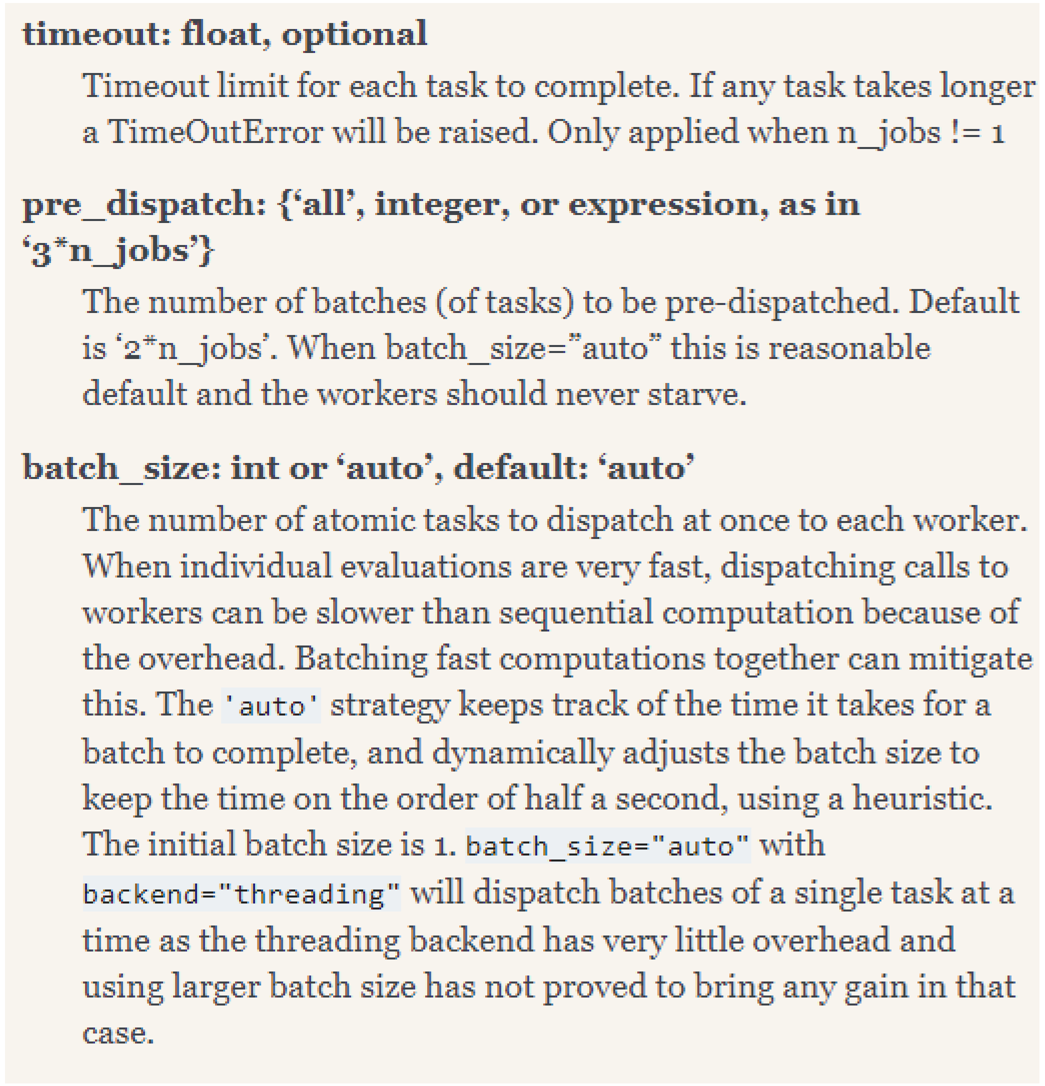
Soft hint to choose the default backend if no specific backend was selected with the parallel_backend context manager. The default process-based backend is 'loky' and the default thread-based backend is 'threading'.

**require: 'sharedmem' or None, default None**

Hard constraint to select the backend. If set to 'sharedmem', the selected backend will be single-host and thread-based even if the user asked for a non-thread based backend with parallel_backend.

**verbose: int, optional**

The verbosity level: if non zero, progress messages are printed. Above 50, the output is sent to stdout. The frequency of the messages increases with the verbosity level. If it more than 10, all iterations are reported.



**timeout: float, optional**

Timeout limit for each task to complete. If any task takes longer a TimeOutError will be raised. Only applied when n_jobs != 1

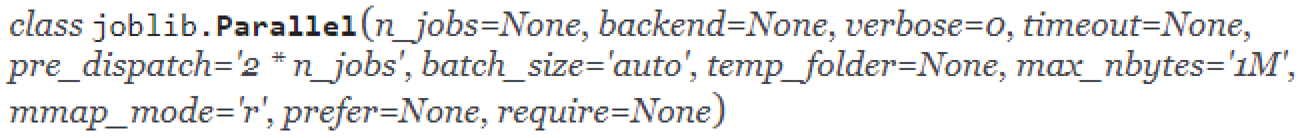
**pre_dispatch: {'all', integer, or expression, as in '3*n_jobs'}**

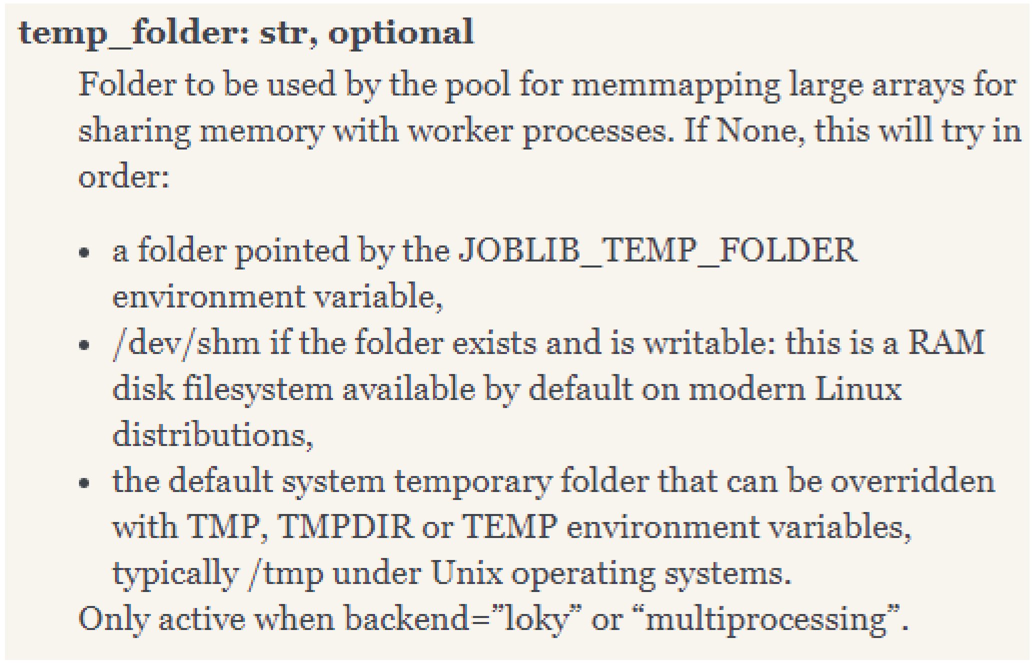
The number of batches (of tasks) to be pre-dispatched. Default is '2*n_jobs'. When batch_size="auto" this is reasonable default and the workers should never starve.

**batch_size: int or 'auto', default: 'auto'**

The number of atomic tasks to dispatch at once to each worker. When individual evaluations are very fast, dispatching calls to workers can be slower than sequential computation because of the overhead. Batching fast computations together can mitigate this. The `'auto'` strategy keeps track of the time it takes for a batch to complete, and dynamically adjusts the batch size to keep the time on the order of half a second, using a heuristic. The initial batch size is 1. `batch_size="auto"` with `backend="threading"` will dispatch batches of a single task at a time as the threading backend has very little overhead and using larger batch size has not proved to bring any gain in that case.

# Joblib Package

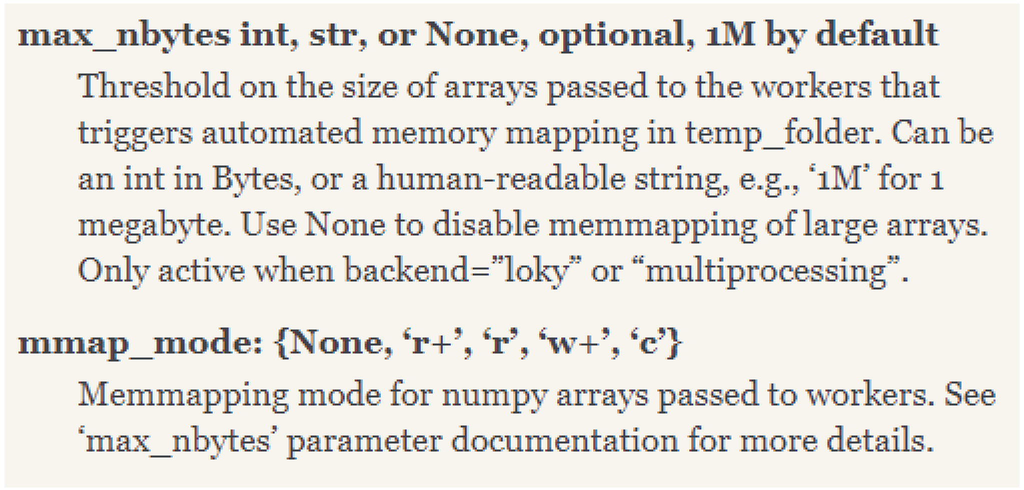

*class* `joblib.Parallel`(*n_jobs=None, backend=None, verbose=0, timeout=None, pre_dispatch='2 * n_jobs', batch_size='auto', temp_folder=None, max_nbytes='1M', mmap_mode='r', prefer=None, require=None*)


**temp_folder: str, optional**

Folder to be used by the pool for memmapping large arrays for sharing memory with worker processes. If None, this will try in order:

- a folder pointed by the JOBLIB_TEMP_FOLDER environment variable,
- /dev/shm if the folder exists and is writable: this is a RAM disk filesystem available by default on modern Linux distributions,
- the default system temporary folder that can be overridden with TMP, TMPDIR or TEMP environment variables, typically /tmp under Unix operating systems.

Only active when backend=”loky” or “multiprocessing”.


**max_nbytes int, str, or None, optional, 1M by default**

Threshold on the size of arrays passed to the workers that triggers automated memory mapping in temp_folder. Can be an int in Bytes, or a human-readable string, e.g., ‘1M’ for 1 megabyte. Use None to disable memmapping of large arrays. Only active when backend=”loky” or “multiprocessing”.

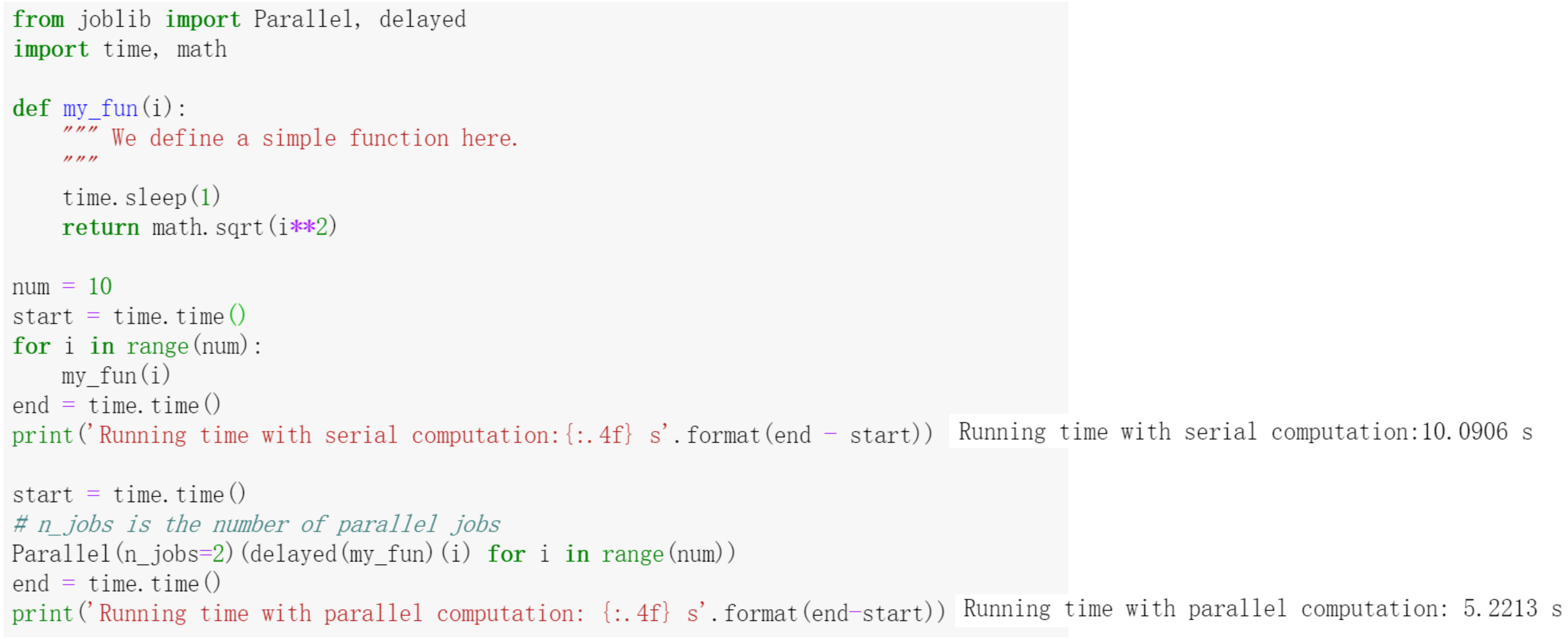
**mmap_mode: {None, ‘r+’, ‘r’, ‘w+’, ‘c’}**

Memmapping mode for numpy arrays passed to workers. See ‘max_nbytes’ parameter documentation for more details.

## Joblib Package

```
from joblib import Parallel, delayed
import time, math

def my_fun(i):
    """ We define a simple function here.
    """
    time.sleep(1)
    return math.sqrt(i**2)

num = 10
start = time.time()
for i in range(num):
    my_fun(i)
end = time.time()
print('{:.4f} s'.format(end - start))

start = time.time()
# n_jobs is the number of parallel jobs
Parallel(n_jobs=2)(delayed(my_fun)(i) for i in range(num))
end = time.time()
print('{:.4f} s'.format(end-start))
```

# Joblib Package



```python
from joblib import Parallel, delayed
import time, math

def my_fun(i):
    """ We define a simple function here.
    """
    time.sleep(1)
    return math.sqrt(i**2)

num = 10
start = time.time()
for i in range(num):
    my_fun(i)
end = time.time()
print('Running time with serial computation:{:.4f} s'.format(end - start))

start = time.time()
# n_jobs is the number of parallel jobs
Parallel(n_jobs=2)(delayed(my_fun)(i) for i in range(num))
end = time.time()
print('Running time with parallel computation: {:.4f} s'.format(end-start))
```

```
Running time with serial computation:10.0906 s
```

```
Running time with parallel computation: 5.2213 s
```

## Multiprocessing(optional)

## Multiprocessing(optional)

Multiprocessing is a package for the Python language which supports the spawning of processes using the API of the standard library’s threading module. multiprocessing has been published in the standard library since python 2.6.

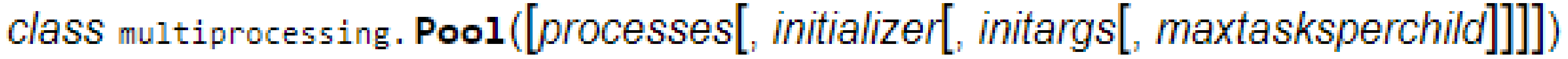
## Multiprocessing(optional)

Features:

- *Objects can be transferred between processes using pipes or multi-producer/multi-consumer queues.*
- *Objects can be shared between processes using a server process or (for simple data) shared memory.*
- *Equivalents of all the synchronization primitives in threading are available.*
- *A Pool class makes it easy to submit tasks to a pool of worker processes.*

# Multiprocessing(optional)

One can create a pool of processes which will carry out tasks submitted to it with the Pool class.

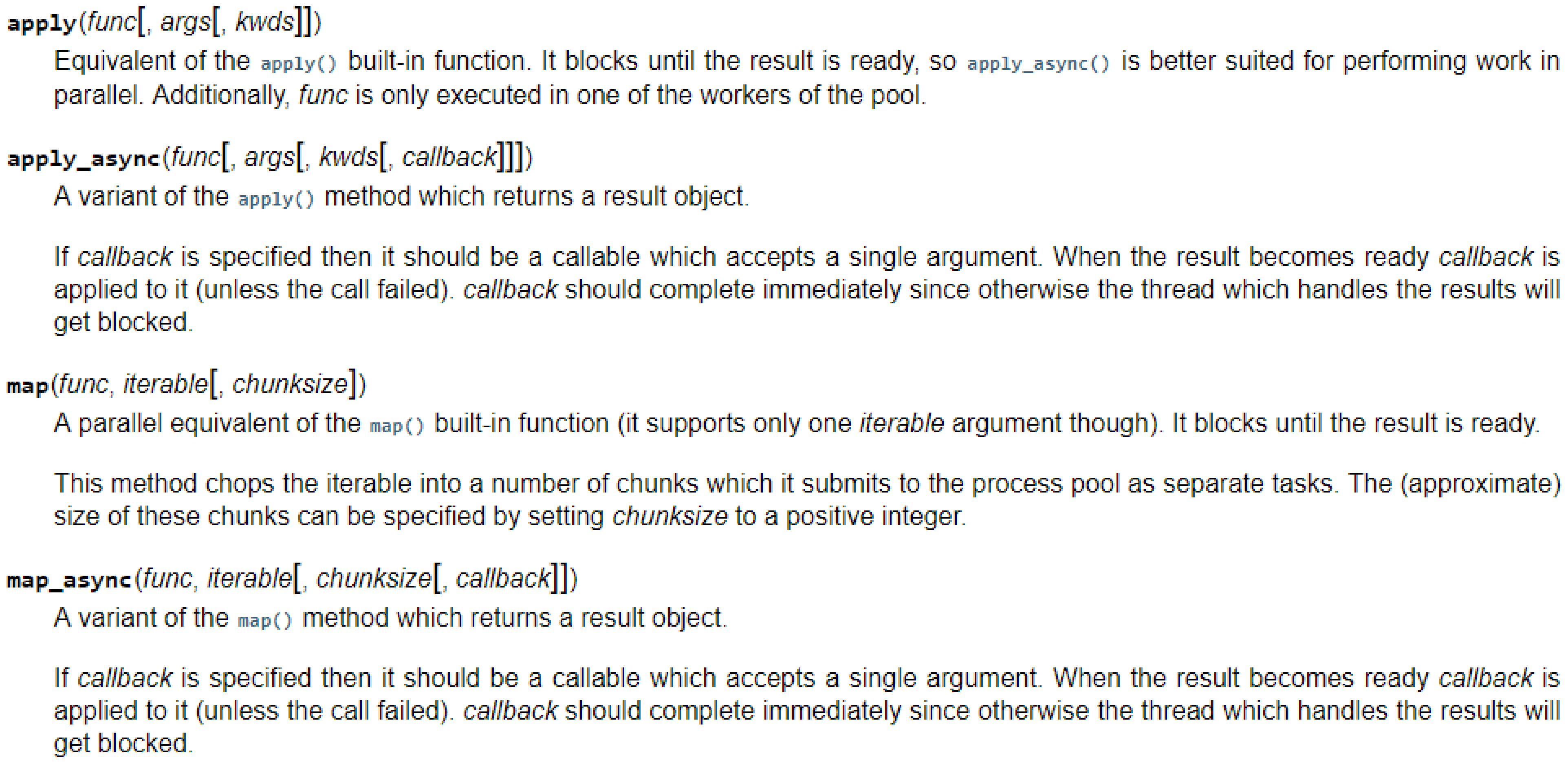


```
class multiprocessing.Pool([processes[, initializer[, initargs[, maxtasksperchild]]]])
```

- A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

- processes is the number of worker processes to use. If processes is None then the number returned by cpu_count() is used. If initializer is not None then each worker process will call initializer(*initargs) when it starts.

# Multiprocessing(optional)



**`apply`**(*func*[, *args*[, *kwds*]])

Equivalent of the `apply()` built-in function. It blocks until the result is ready, so `apply_async()` is better suited for performing work in parallel. Additionally, *func* is only executed in one of the workers of the pool.

**`apply_async`**(*func*[, *args*[, *kwds*[, *callback*]]])

A variant of the `apply()` method which returns a result object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it (unless the call failed). *callback* should complete immediately since otherwise the thread which handles the results will get blocked.

**`map`**(*func*, *iterable*[, *chunksize*])

A parallel equivalent of the `map()` built-in function (it supports only one *iterable* argument though). It blocks until the result is ready.

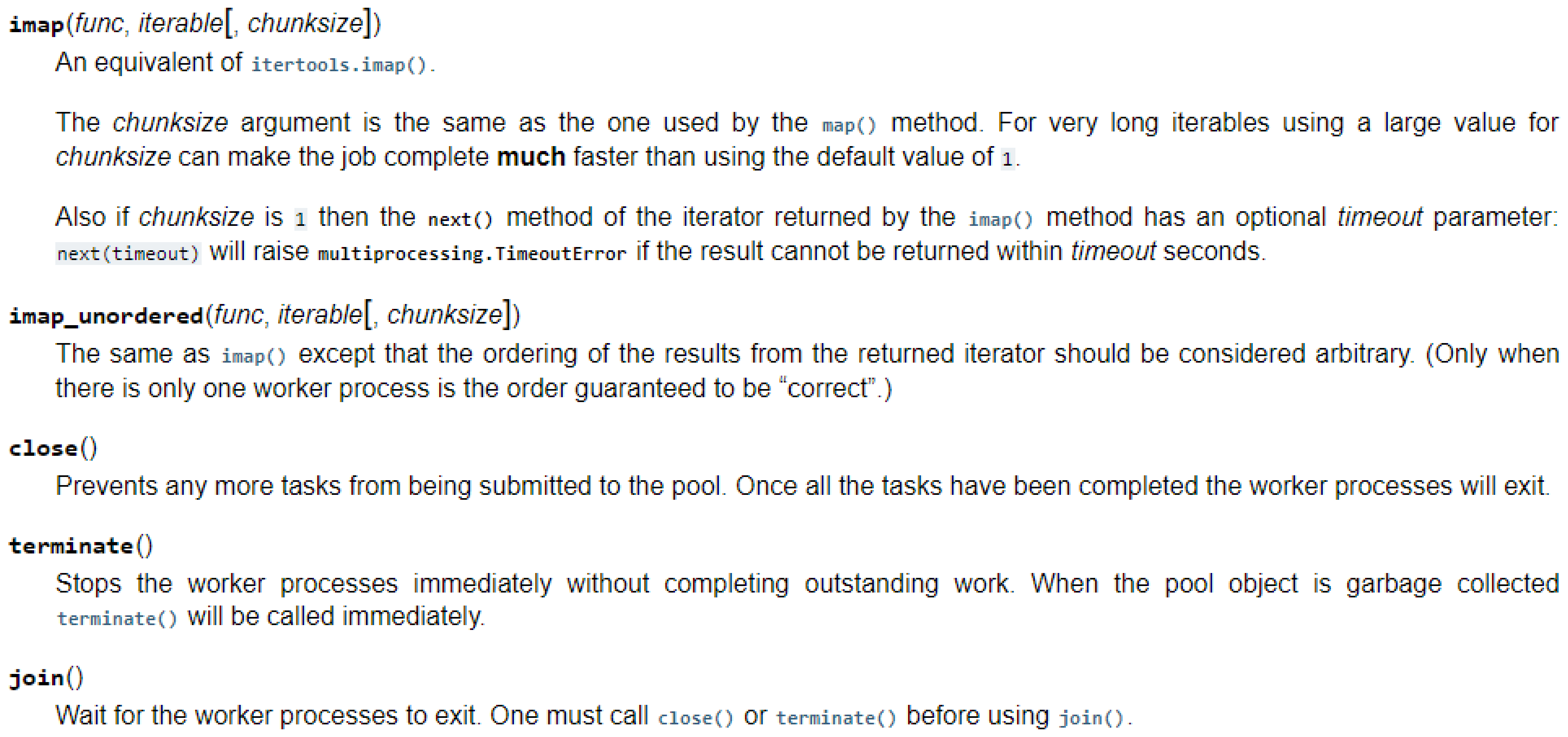
This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer.

**`map_async`**(*func*, *iterable*[, *chunksize*[, *callback*]])

A variant of the `map()` method which returns a result object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it (unless the call failed). *callback* should complete immediately since otherwise the thread which handles the results will get blocked.

# Multiprocessing(optional)



**`imap`**(*func*, *iterable*[, *chunksize*])

An equivalent of `itertools.imap()`.

The *chunksize* argument is the same as the one used by the `map()` method. For very long iterables using a large value for *chunksize* can make the job complete **much** faster than using the default value of `1`.

Also if *chunksize* is `1` then the `next()` method of the iterator returned by the `imap()` method has an optional *timeout* parameter: `next(timeout)` will raise `multiprocessing.TimeoutError` if the result cannot be returned within *timeout* seconds.

**`imap_unordered`**(*func*, *iterable*[, *chunksize*])

The same as `imap()` except that the ordering of the results from the returned iterator should be considered arbitrary. (Only when there is only one worker process is the order guaranteed to be "correct".)

**`close`**()

Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.

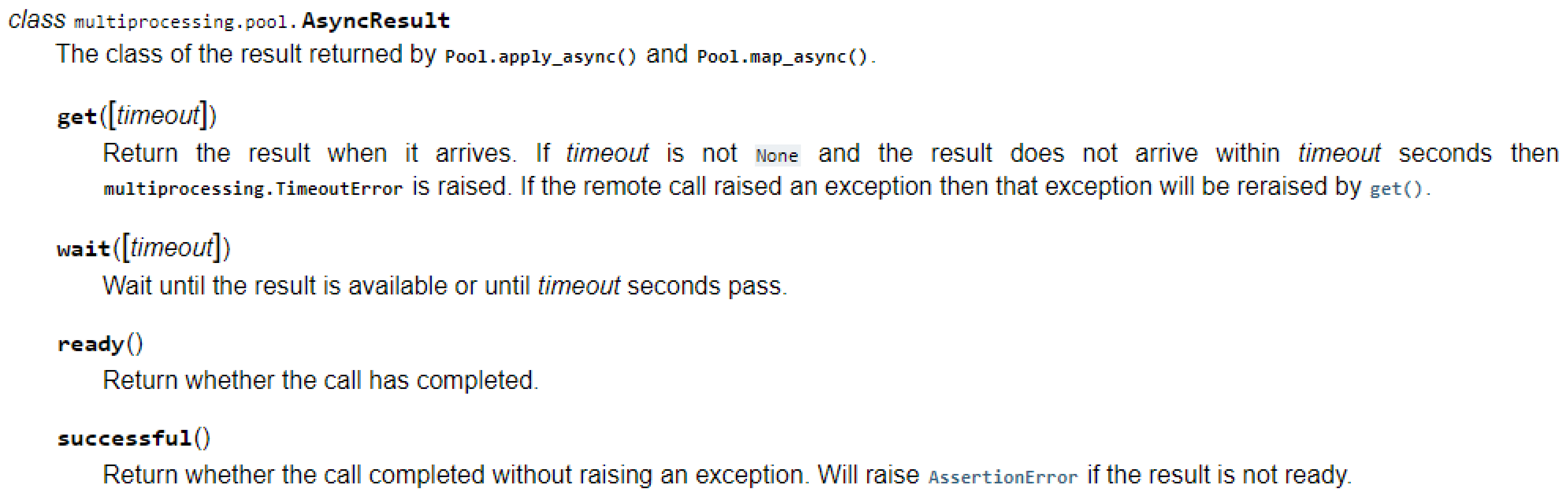
**`terminate`**()

Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected `terminate()` will be called immediately.

**`join`**()

Wait for the worker processes to exit. One must call `close()` or `terminate()` before using `join()`.

# Multiprocessing(optional)



*class* `multiprocessing.pool.`**`AsyncResult`**

The class of the result returned by `Pool.apply_async()` and `Pool.map_async()`.

**`get`**([*timeout*])

Return the result when it arrives. If *timeout* is not `None` and the result does not arrive within *timeout* seconds then `multiprocessing.TimeoutError` is raised. If the remote call raised an exception then that exception will be reraised by `get()`.

**`wait`**([*timeout*])

Wait until the result is available or until *timeout* seconds pass.

**`ready`**()

Return whether the call has completed.

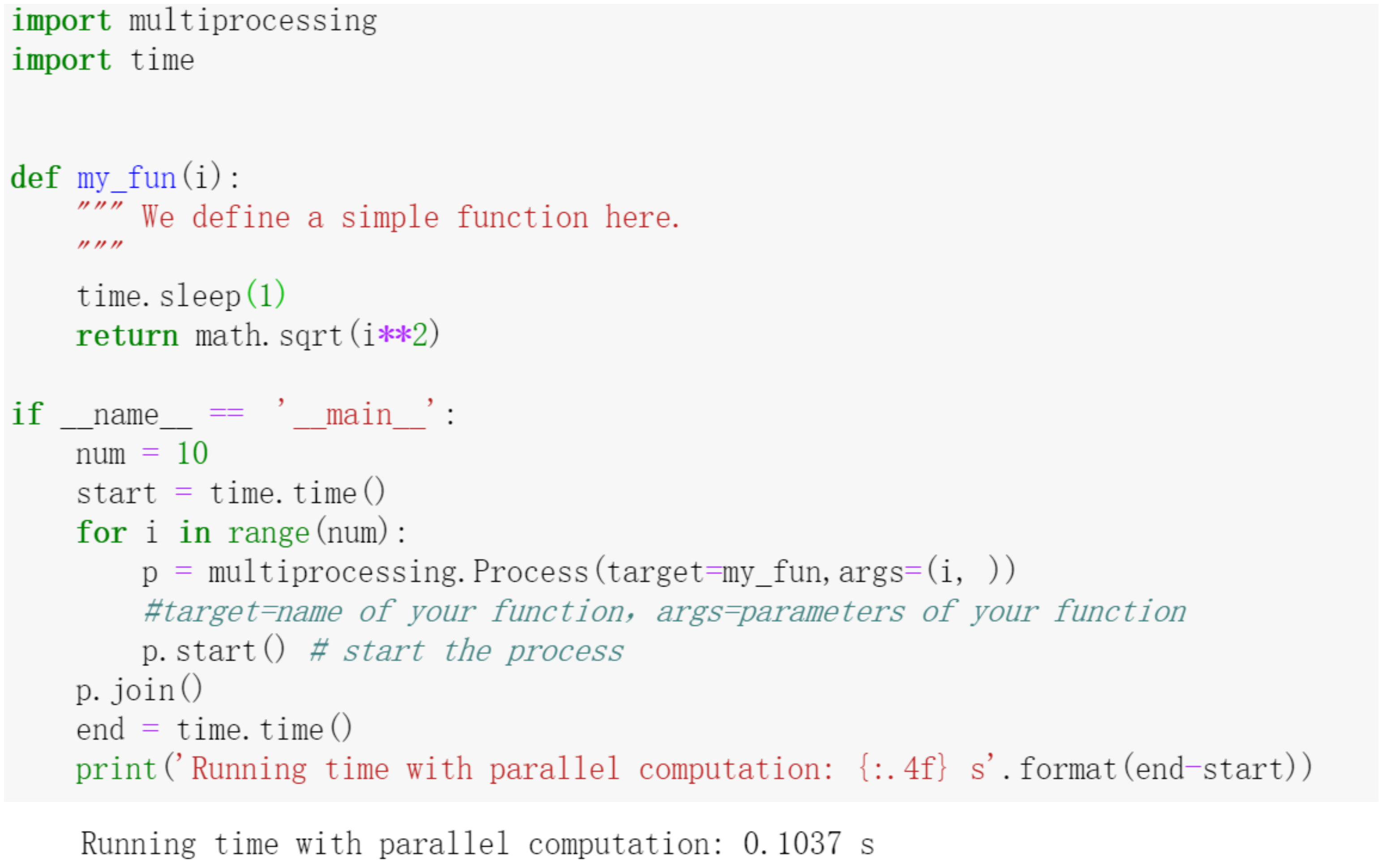
**`successful`**()

Return whether the call completed without raising an exception. Will raise `AssertionError` if the result is not ready.

# Multiprocessing(optional)

```
import multiprocessing
import time


def my_fun(i):
    """ We define a simple function here.
    """
    time.sleep(1)
    return math.sqrt(i**2)

if __name__ ==  '__main__':
    num = 10
    start = time.time()
    for i in range(num):
        p = multiprocessing.Process(target=my_fun,args=(i, ))
        #target=name of your function，  args=parameters of your function
        p.start() # start the process
    p.join()
    end = time.time()
    print('Running time with parallel computation: {:.4f} s'.format(end-start))
```

# Multiprocessing(optional)



```
import multiprocessing
import time


def my_fun(i):
    """ We define a simple function here.
    """
    time.sleep(1)
    return math.sqrt(i**2)

if __name__ == '__main__':
    num = 10
    start = time.time()
    for i in range(num):
        p = multiprocessing.Process(target=my_fun,args=(i, ))
        #target=name of your function, args=parameters of your function
        p.start() # start the process
    p.join()
    end = time.time()
    print('Running time with parallel computation: {:.4f} s'.format(end-start))
```

```
Running time with parallel computation: 0.1037 s
```