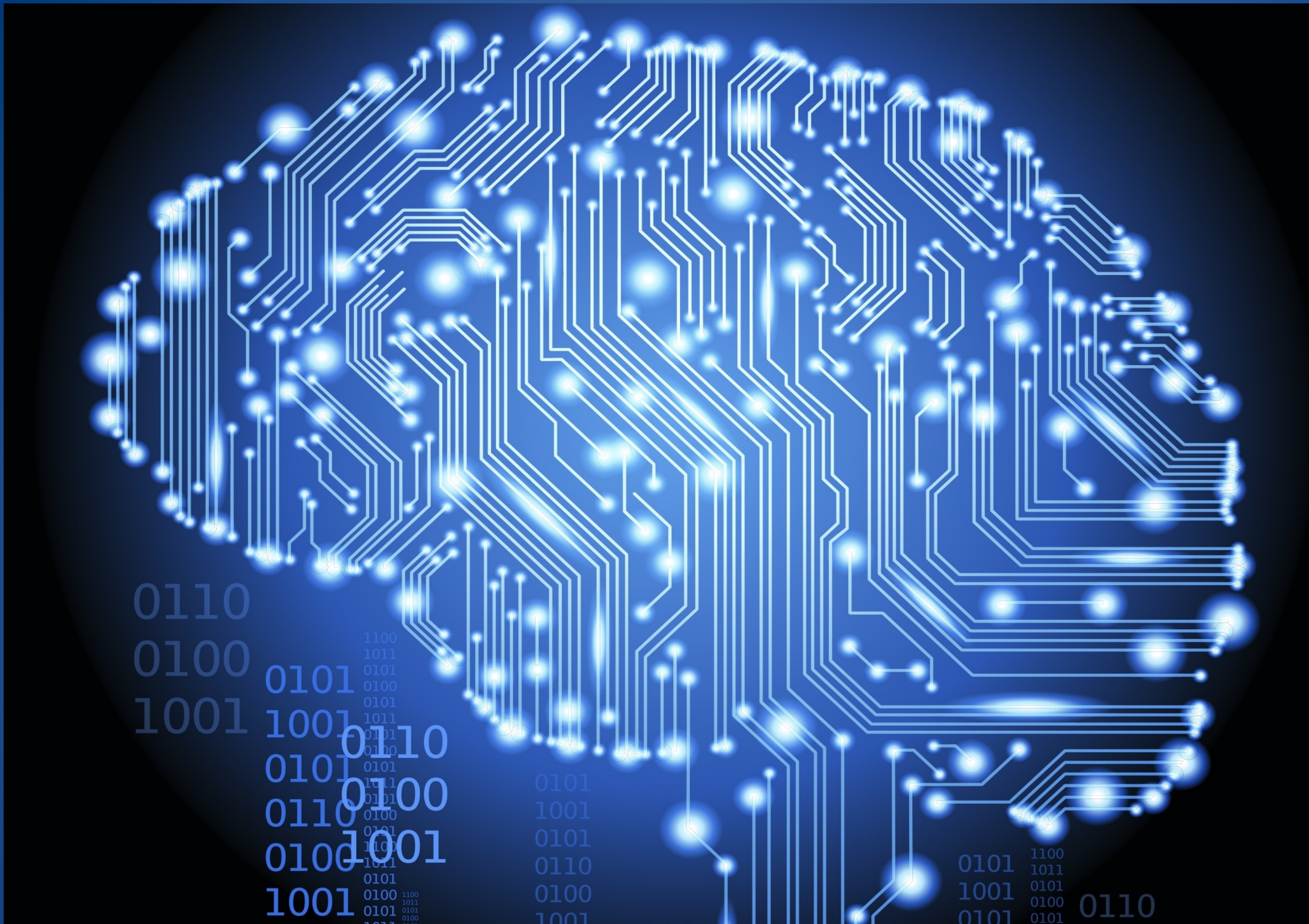


Ekaterina Tuzova

Numpy: Vectorize your brain

K nearest neighbors

```
def knn(X_train, y_train, X_test, k, dist):
    n_classes = len(set(y_train))
    y_test = []
    for i in range(0, len(X_test)):
        distances = []
        for j in range(0, len(X_train)):
            distances.append(dist(X_train[j], X_test[i]))
        nearest = sorted(zip(distances, y_train))[:k]
        nearest_by_class = [(len([x for x in nearest
                if x[1] == i]), i) for i in range(1, n_classes + 1)]
        y_test.append(max(nearest_by_class)[1])
    return y_test
```

https://archive.ics.uci.edu/ml/datasets/Wine

NumPy

What is NumPy?

Numpy is the fundamental package for scientific computing with Python.

IPython

Python and Performance

Python is fast

Python is slow

Euclidian distance

```
import math
def euclidean(xs, ys):
    n = len(xs) # == len(ys)
    acc = 0.
    for i in range(n):
        acc += (xs[i] - ys[i]) ** 2
    return math.sqrt(acc)
```

“Magic” timeit

```
import random
def setup(size):
    xs = [random.random() for _ in range(size)]
    ys = [random.random() for _ in range(size)]
    return xs, ys
```

```
%%timeit xs, ys = setup(8192)
euclidean(xs, ys)
```

```
100 loops, best of 3: 2.67 ms per loop
```

Euclidian distance. C

```
%load_ext biteymagic
```

```
%%bitey
double euclideanDistance(double x[3], double y[3])
{
 double Sum;
 double distance;
 for(int i=0;i<3;i++)
 {
   Sum = Sum + pow((x[i]-y[i]),2.0);
   distance = sqrt(Sum);
 }
 return distance;
}
```

Euclidian distance. C

```
%load_ext biteymagic
```

```
%%bitey
double euclideanDistance(double x[3], double y[3])
{
 double Sum;
 double distance;
 for(int i=0;i<3;i++)
 {
   Sum = Sum + pow((x[i]-y[i]),2.0);
   distance = sqrt(Sum);
 }
 return distance;
}
```

```
%%timeit xs, ys = setup(8192)
euclideanDistance(xs, ys)
```

```
10000 loops, best of 3: 28 µs per loop
```

Euclidian distance

```python
import math
def euclidean(xs, ys):
    n = len(xs) # == len(ys)
    acc = 0.
    for i in range(n):
        acc += (xs[i] - ys[i]) ** 2
    return math.sqrt(acc)
```

line_profiler and “magic” lprun

```
%load_ext line_profiler
```

```
%lprun -f euclidean euclidean(*setup(8192))
```

Euclidian distance

```
Timer unit: 1e-06 s

Total time: 0.015907 s
File: <ipython-input-1-51b5d0f5e2ad>
Function: euclidean at line 2

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           def euclidean(xs, ys):
     3         1            3      3.0      0.0      n = len(xs) # == len(ys)
     4         1            1      1.0      0.0      acc = 0.
     5      8193         6037      0.7     38.0      for i in range(n):
     6      8192         9861      1.2     62.0          acc += (xs[i] - ys[i]) ** 2
     7         1            5      5.0      0.0      return math.sqrt(acc)
```

Compiled languages

Interpreted languages

What can be done?

NumPy

Ufuncs

Universal **func**tions

Special type of function defined within a numpy library and it operate element-wise on arrays.

Arithmetic operations

```
a = range(4)
b = [value + 1 for value in a]
print(b)
```

```
[1, 2, 3, 4]
```

Arithmetic operations

```
a = range(4)
b = [value + 1 for value in a]
print(b)
```

```
[1, 2, 3, 4]
```

```
import numpy as np
```

```
a = np.arange(4)
b = a + 1
print(b)
```

```
[1 2 3 4]
```

Arithmetic operations

```
a = np.arange(4)
b = np.full(4, 2)
```

```
a*b
```

```
array([ 0.,  2.,  4.,  6.])
```

Arithmetic operations

```
%%timeit a = np.arange(100000)
a + 1
```

10000 loops, best of 3: 108 µs per loop

```
%%timeit a = range(100000)
[value + 1 for value in a]
```

100 loops, best of 3: 8.96 ms per loop

Ufuncs available

- Arithmetic
- Bitwise
- Comparison
- Trigonometric
- Floating

...

Slicing and indexing

Slicing and indexing

```python
x = np.arange(4)
```

```python
x[:2]
```

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

Slicing and indexing

```
x = np.arange(4)
```

```
x[:2]
```

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

```
y = x[1:]
y[0] = 42
print(x)
```

```
[ 0 42  2  3]
```

Multidimensional arrays

```
X = np.arange(6).reshape((2, 3))
X
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
X[0, 1]
```

```
1
```

```
X[:1, 1:]
```

```
array([[1, 2]])
```

Multidimensional arrays

```
X = np.arange(6).reshape((2, 3))
X
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
X[:, 1]  # X.T[1]
```

```
array([1, 4])
```

```
X[0, :] # X[0]
```

```
array([0, 1, 2])
```

Index arrays

```
y = x[[2,0,1]]
y
```

```
array([ 2,  0, 42])
```

Index arrays

```
y = x[[2,0,1]]
y
```

```
array([ 2,  0, 42])
```

```
y[0] = 1
```

Index arrays

```
y = x[[2,0,1]]
y
```

```
array([ 2,  0, 42])
```

```
y[0] = 1
```

```
x
```

```
array([ 0, 42,  2,  3])
```

Masking

```
x
```

```
array([ 0, 42,  2,  3])
```

```
x[np.array([False, True, True, True])]
```

```
array([42,  2,  3])
```

Masking

```
x
```

```
array([ 0, 42,  2,  3])
```

```
x[np.array([False, True, True, True])]
```

```
array([42,  2,  3])
```

```
x[x >= 2]
```

```
array([42,  2,  3])
```

Test train split

```
def train_test_split(X, y, ratio):
    mask = np.random.random(len(y)) < ratio
    return X[mask], y[mask], X[~mask], y[~mask]
```

Test train split

```
def train_test_split(X, y, ratio):
    X_train = []
    y_train = []
    X_test = []
    y_test = []

    numbers = [i for i in range(len(X))]
    shuffle(numbers)
    numbers = numbers[0: int(ratio * len(X))]

    for i in range(len(X)):
        if i in numbers:
            X_train.append(X[i])
            y_train.append(y[i])
        else:
            X_test.append(X[i])
            y_test.append(y[i])

    return X_train, y_train, X_test, y_test
```

Broadcasting

Broadcasting

Broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations.

Broadcasting rules

1. If two arrays differ in their number of dimension, the shape of the array with the fewer dimensions is padded with ones on it's leading(left) size.

2. If the shape of two arrays doesn't match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

3. If these conditions are not met, raise a ValueError: operands could not be broadcast together with shapes

Broadcasting. Example

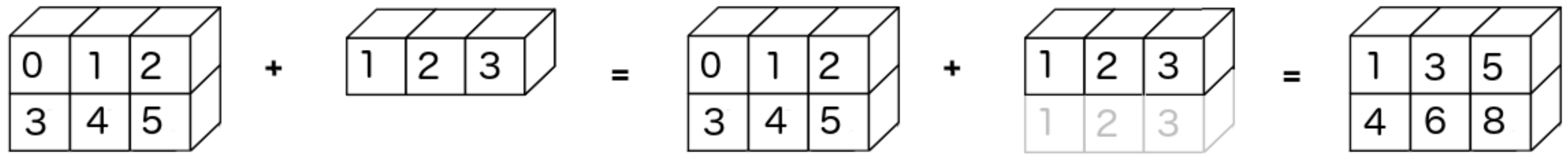
```
X = np.arange(6).reshape((2, 3))
X
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
Y = [[1, 2, 3]]
```

```
X + Y
```

```
array([[1, 3, 5],
       [4, 6, 8]])
```



np.newaxis

```
X = np.arange(6).reshape((3, 2))
X
```

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
X + np.array([1, 2, 3])
```

np.newaxis

```
X = np.arange(6).reshape((3, 2))
X
```

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
X + np.array([1, 2, 3])
```

```
ValueError: operands could not be broadcast together with shapes
(3,2) (3,)
```

np.newaxis

```
X = np.arange(6).reshape((3, 2))
X
```

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
Y = np.array([1, 2, 3])[:, np.newaxis]
Y.shape
```

```
(3, 1)
```

```
X + Y
```

```
array([[1, 2],
       [4, 5],
       [7, 8]])
```

Aggregations

Aggregations

```
X = np.arange(6).reshape((2, 3))
```

```
X.mean()
```

```
2.5
```

Aggregations

```
X = np.arange(6).reshape((2, 3))
```

```
X.mean()
```

```
2.5
```

```
X.mean(axis=0)
```

```
array([ 1.5,  2.5,  3.5])
```

```
X.mean(axis=1)
```

```
array([ 1.,  4.])
```

NumPy resume

Basic ideas to make you code faster:

- Ufuncs
- Slicing and indexing
- Broadcasting
- Aggregations

k-means

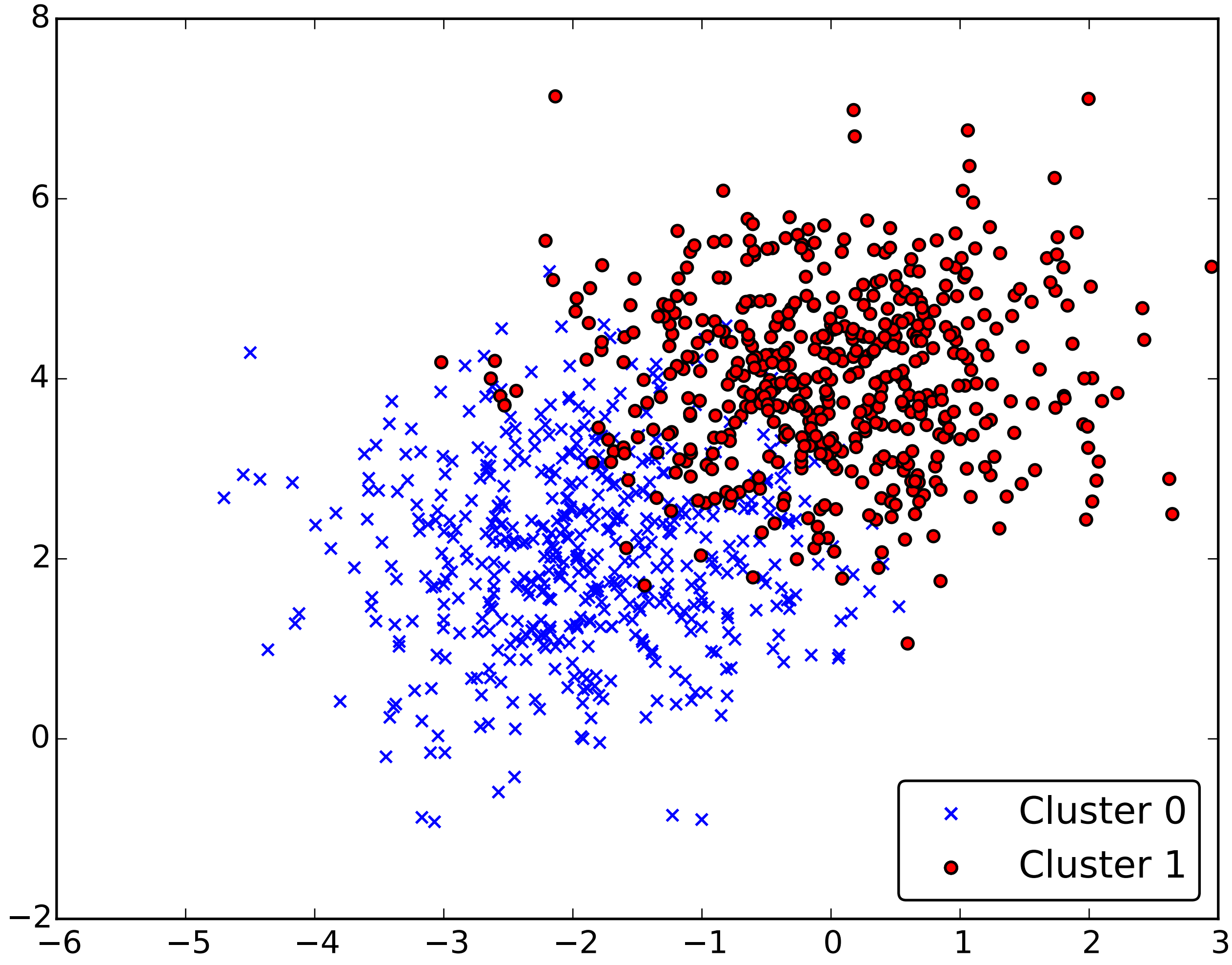
Algorithm

1. Clusters the data into k groups where k is predefined.
2. Select k points at random as cluster centers.
3. Assign objects to their closest cluster center according to the Euclidean distance function.
4. Calculate the centroid or mean of all objects in each cluster.
5. Repeat steps 2, 3 and 4 until the same points are assigned to each cluster in consecutive rounds.

Synthetic data

```
import numpy as np
from numpy import random
def sample(size, ratio=.5):
    y = np.random.random(size) <= ratio
    n1 = np.count_nonzero(y)
    n0 = size - n1

    covar = np.diag([1, 1])
    X = np.empty((size, 2))
    X[y == 0, :] = random.multivariate_normal([-2, 2], covar, n0)
    X[y == 1, :] = random.multivariate_normal([0, 4], covar, n1)
    return X, y
```



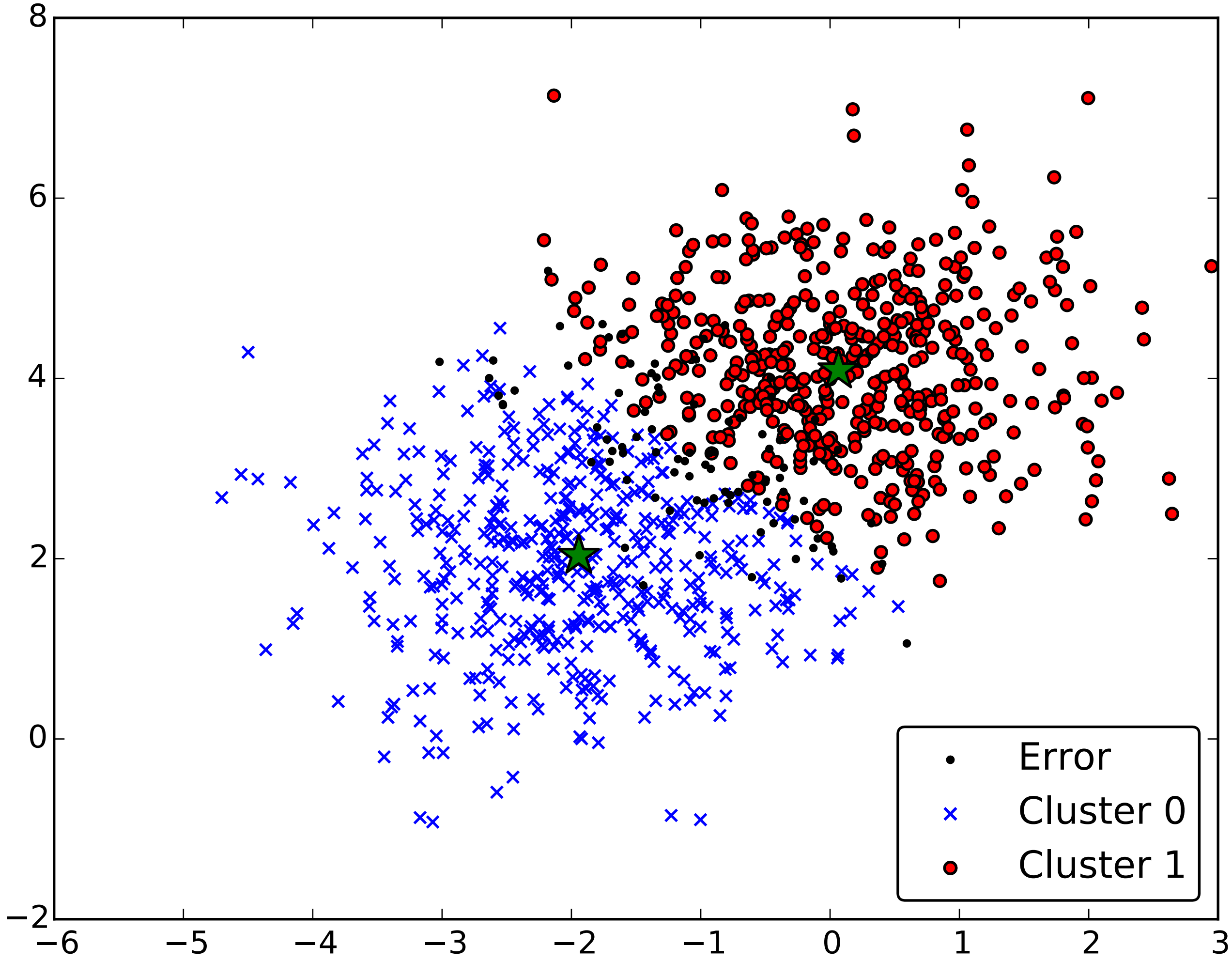
8
6
4
2
0
−2
−6
−5
−4
−3
−2
−1
0
1
2
3
Cluster 0
Cluster 1

Vectorized euclidian distance

```
def ceuclidean(A, B):
    assert A.ndim == B.ndim == 2
    D = np.empty((len(A), len(B)), dtype=np.float64)
    for i, Ai in enumerate(A):
        D[i, :] = np.sqrt(np.square(Ai - B).sum(axis=1))
    return D
```

k-means

```
def kmeans(X, n_clusters):
    centers = init_centers(X, n_clusters)
    y = None
    while True:
        D = ceuclidean(centers, X)
        new_y = D.argmin(axis=0)
        if np.array_equal(y, new_y):
            break
        y = new_y
        for i in range(n_clusters):
            centers[i] = X[y == i].mean(axis=0)
    return centers, y
```


8
6
4
2
0
−2
−6
−5
−4
−3
−2
−1
0
1
2
3
Error
Cluster 0
Cluster 1

Thank you.

@ktisha