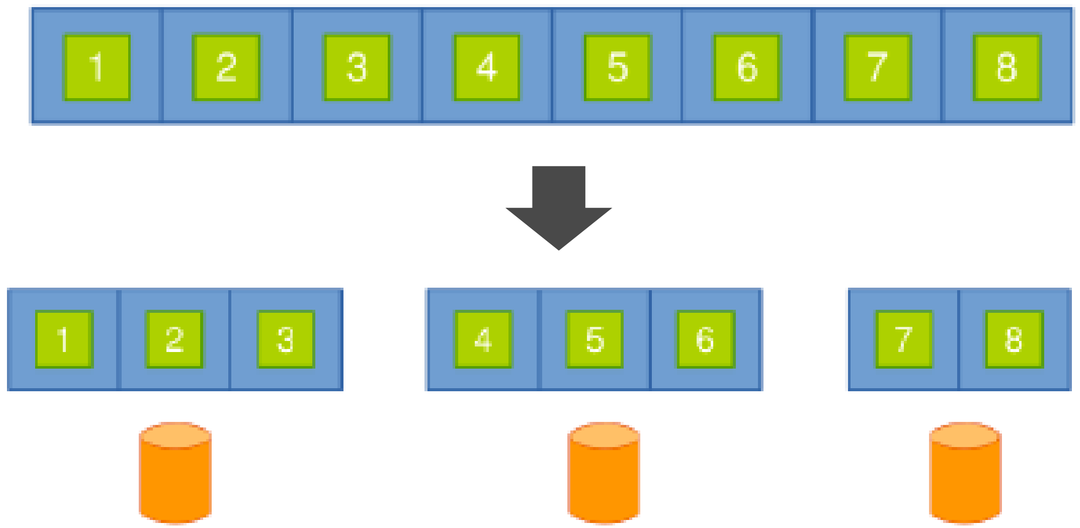
Practice 3

K-Nearest Neighbor

Problem

- Multi-threading problem: predict MNIST dataset's label using K-Nearest Neighbors
- Use "--master local" argument *to select number of **N** cores and execute your pythonscript.py*
  - spark-submit --master local[*N*] YOUR_PYTHON_SCRIPT.py



Dataset for KNN

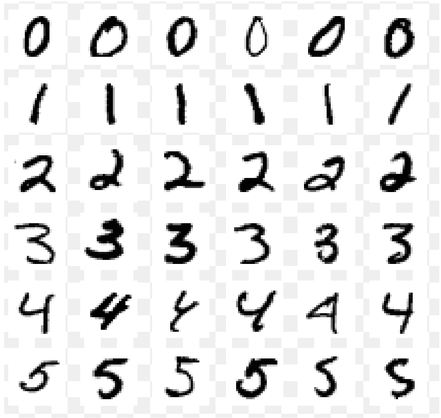
➢ **MNIST : Recognition of handwritten digits**

- There are 10 handwritten digits(0~9) in bitmap format.

➢ 784 Features (28 x 28 pixel values)

1. Pixel 1
2. Pixel 2
…
…
…
783. Pixel 783
784. Pixel 784

* The MNIST Database :



➢ **You can download dataset using *sklearn.datsets.fetch_openml* library**

https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_openml.html

Practice 3

1. **Compare processing time for classification when you use Multi-threading or single-threading**

   ※ A few minutes will be needed for loading large dataset

   - You can use only one core with "**spark-submit --master local PYTHON_SCRIPT.py**" command
   - Or, maximum number of cores with "**local[*]**"

2. **Use predefined classes in *sklearn.neighbors.KNeighborClassifier***

   Parameters for the method

   - **n_neighbors: 11 (Don't change the other parameters)**

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Practice 3

3. How to train the model using RDD data format

- Before training the model, you need to save data into your memory using ***cache()*** function.
- For example

```
trRDDs.cache()
tsRDDs.cache()
```

- In this example, ***trRDDs***: training data points & ***tsRDDs***: test data points

- Then, you can easily train KNN model provided by scikit-learn using ***fit()*** function
- For example

```
Knn = KNN(n_neighbors = K).fit(trRDDs.collect(), trLabel)
```

- In this example, ***n_neighbors***: number of neighbors to use for kneighbors queries & ***trLabel***: label of training data points, ***collect()***: Return all the elements of datasets as an array at the driver program.

https://spark.apache.org/docs/latest/rdd-programming-guide.html
https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Practice 3

4. **After training the models, get the accuracy & F1 score for each label using test data points**
5. **You need to use predefined arguments we suggest.**
   - **Number of train data points: 30,000**

     Use first thirty thousands(30,000) data points as training datasets

   - **Number of test data points: 10,000**

     Use next ten thousands(10,000) data points as test datasets

   - **Number of partitions: 500**

     You can split data when you make it RDDs.

     For example, “ ***RDD = sc.parallelize(Data, numPartition)*** ”

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Submission

- **You need to submit two files(result.txt and time.txt)**
  - **result.txt**

    Write accuracy score of KNN result, using ***sklearn.metrics.accuracy_score*** library

    Then, write F1 score of KNN result, using ***sklearn.metrics.f1_score*** library

    When you calculate F1 score, you need to use parameter **average = 'macro'**
  - **time.txt**

    Write time difference, when you use multi-threading(full thread) and single-threading

```
accuracy : 0.9580
f1score: 0.9578
```

```
multi-threading time: 498.9002
single-threading time: 1816.3231
```

Windows

```
accuracy : 0.9580
f1score: 0.9578
```

```
multi-threading time: 569.6994
single-threading time: 1528.6397
```

Linux

Solution

➢ load MNIST dataset and libraries for KNN & metrics

```
import time
import numpy as np

from sklearn.datasets import fetch_openml
from sklearn.metrics import accuracy_score, f1_score
from sklearn.neighbors import KNeighborsClassifier as KNN

from pyspark import SparkConf, SparkContext
```

Library for loading MNIST data

Metrics for accuracy & f1score

➢ Set the parameters

```
K = 11
numTrain = 30000
numTest = 10000
numTotal = numTrain + numTest

# If you get JAVA out of memory(OOM),
# then you can solve the problem with increasing numPartition
numPartition = 500
```

- # of partition to split the data
- If you get JAVA OOM, then increase your number of partition
- But it takes more and more time to run the code

Solution

➢ Load data

```
def LOAD_DATA(data):
    print("Loading {} dataset".format(data))
    mnist = fetch_openml(data)
    print("Successfully load data")
    return mnist
```

- Using “fetch_openml” library, you can load mnist dataset(type: dataframe)
- But it takes a few time.

```
# Load data
mnist = LOAD_DATA('mnist_784')
data = mnist.data[:numTotal]
target = mnist.target[:numTotal]
```

We use part of data, since this data is very heavy

➢ Initialize a SparkContext

```
start = time.time()

conf = SparkConf()
sc = SparkContext(conf=conf)
```

Solution

➢ Transform data into Spark RDDs

Split data into train, test data points

```
trData, tsData = data[:numTrain], data[numTrain:numTotal]
trLabel, tsLabel = target[:numTrain], target[numTrain:numTotal]
```

```
trRDDs = sc.parallelize(trData.tolist(), numPartition)
tsRDDs = sc.parallelize(tsData.tolist(), numPartition)
```

- Transform data into Spark RDDs.
- Don’t forget “numPartition”
- sc.parallelize() function gets list type of data

➢ Caching the data into memory

```
trRDDs.cache()
tsRDDs.cache()
```

Since most RDD operations are lazy, we need to save our data into memory. Unless you do, then your machine load data from the beginning whenever you need.

Solution

Train the model using training RDDs

- Train the model with training data points

```
Knn = KNN(n_neighbors = K).fit(trRDDs.collect(), trLabel)
Knn = sc.broadcast(Knn)
```

Share the model

- Predict label of test data points & Save the results

```
results = tsRDDs.map(lambda x:Knn.value.predict(np.array(x).reshape(1,-1)))
results = results.collect()
```

We can write ".value" to use broadcasted model or variable

- Transform data type as list

```
prediction = [int(x[0]) for x in results]
real = [int(x[0]) for x in tsLabel]
```

Solution

- Get accuracy and F1 score

```
accuracy = accuracy_score(real, prediction)
f1score = f1_score(real, prediction, average='macro')
f = open("result.txt",'w')
f.write("accuracy : {:.4f}\n".format(accuracy))
f.write("f1score: {:.4f}".format(f1score))

sc.stop()

end = time.time()

g = open("time.txt",'w')
g.write("time: {:.4f}".format(end-start))
```

Since Mnist has multiple labels, we need to use average='macro'

Solution

➢ **Result**

- Your results might be like the following
- But depending on your computer spec, running time will be different a little bit

```
accuracy : 0.9580
f1score: 0.9578
```

```
multi-threading time: 498.9002
single-threading time: 1816.3231
```

Windows

```
accuracy : 0.9580
f1score: 0.9578
```

```
multi-threading time: 569.6994
single-threading time: 1528.6397
```

Linux