HW1

1. Iterative K-means clustering on Spark

(1) L1 Distance

The code for this part is shown as below

```python
import operator
import sys
from io import BytesIO
from pyspark import SparkConf, SparkContext
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg

# Macros.
MAX_ITER = 20
DATA_PATH = "gs://6893_bucket_1/HW1/data.txt"
C1_PATH = "gs://6893_bucket_1/HW1/c1.txt"
C2_PATH = "gs://6893_bucket_1/HW1/c2.txt"
NORM = 2


# Helper functions.
def closest(p, centroids, norm):
    """
    Compute closest centroid for a given point.
    Args:
```

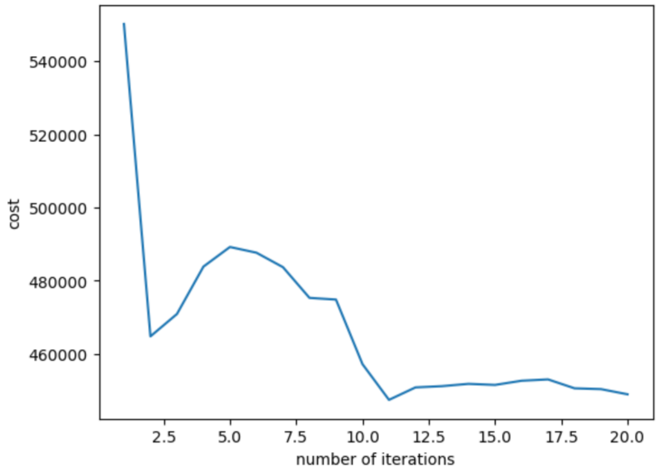
```python

# Helper functions.
def closest(p, centroids, norm):
    """
    Compute closest centroid for a given point.
    Args:
        p (numpy.ndarray): input point
        centroids (list): A list of centroids points
        norm (int): 1 or 2
    Returns:
        int: The index of closest centroid.
    """
    closest_c = min([(i, linalg.norm(p - c, norm))
                     for i, c in enumerate(centroids)],
                    key=operator.itemgetter(1))[0]
    return closest_c


# K-means clustering
def kmeans(data, centroids, norm=1):
    """
```

```python
    # iterative k-means
    costs=[]
    for _ in range(MAX_ITER):
        # Transform each point to a combo of point, closest centroid, count=1
        # point -> (closest_centroid, (point, 1))
        combo = data.map(lambda point: (closest(point, centroids, norm), (point, 1)))

        cost = combo.map(lambda x: linalg.norm(x[1][0] - centroids[x[0]], norm)).sum()
        costs.append(cost)

        # Re-compute cluster center
        # For each cluster center (key), aggregate its values
        # by summing up points and count
        aggregated_combo = combo.reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))

        # Average the points for each centroid: divide sum of points by count
        updated_centroids = aggregated_combo.mapValues(lambda value: value[0] / value[1])
        # Use collect() to turn RDD into list
        centroids = updated_centroids.map(lambda x: x[1]).collect()

    out = combo.map(lambda x: (x[0],x[1][0]))
    print(costs)

    return costs, centroids, out

```

```python
def main():
    # Spark settings
    conf = SparkConf()
    sc = SparkContext(conf=conf)

    # Load the data, cache this since we're accessing this each iteration
    data = sc.textFile(DATA_PATH).map(
            lambda line: np.array([float(x) for x in line.split()])
            ).cache()

    # Load the initial centroids c1, split into a list of np arrays
    centroids1 = sc.textFile(C1_PATH).map(
            lambda line: np.array([float(x) for x in line.split(' ')])
            ).collect()

    # Load the initial centroids c2, split into a list of np arrays
    centroids2 = sc.textFile(C2_PATH).map(
            lambda line: np.array([float(x) for x in line.split(' ')])
            ).collect()

    costs1=kmeans(data, centroids1)
    plt.plot(range(1, MAX_ITER + 1), costs1, marker='o')
    plt.xlabel('Number of Iterations')
    plt.ylabel('Cost')
    plt.title('Cost vs. Iterations')
    plt.show()

```

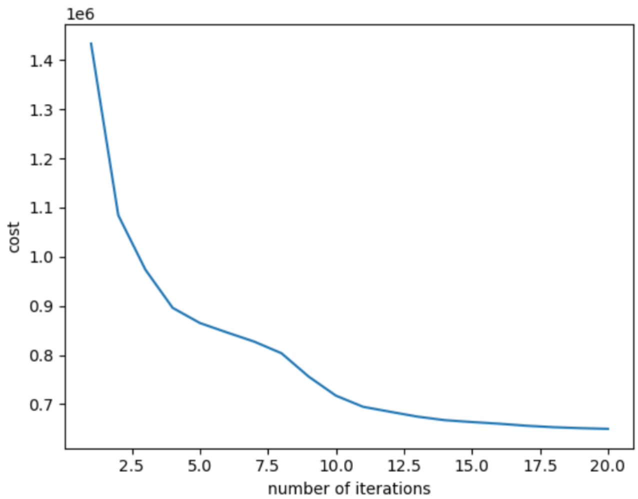
```python


    costs2 = kmeans(data, centroids2)
    plt.plot(range(1, MAX_ITER + 1), costs2, marker='o')
    plt.xlabel('Number of Iterations')
    plt.ylabel('Cost')
    plt.title('Cost vs. Iterations')
    buffer = BytesIO()
    plt.savefig(buffer, format='png')


    sc.stop()



    # TODO: Run the kmeans clustering and complete the HW

if __name__ == "__main__":
    main()

```

The plot of the costs for “c1” is:



The plot of the costs for ‘’c2’ is:

```
In [16]: if __name__ == "__main__":
             main()
```



(2) L2 distance

Modify the line in (1):

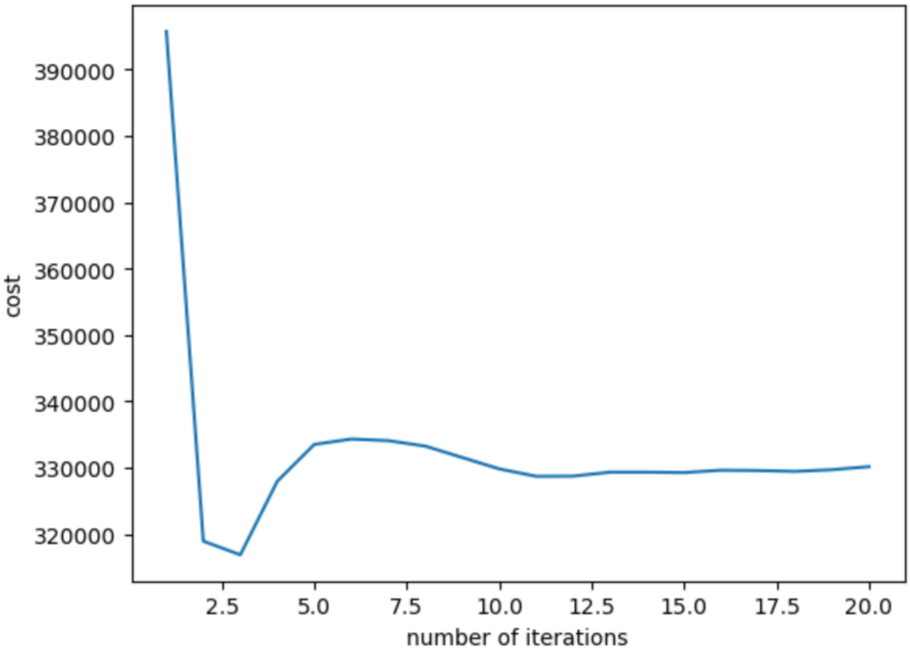
```
def kmeans(data, centroids, norm=1):
```

to:

```
def kmeans(data, centroids, norm=2):
```

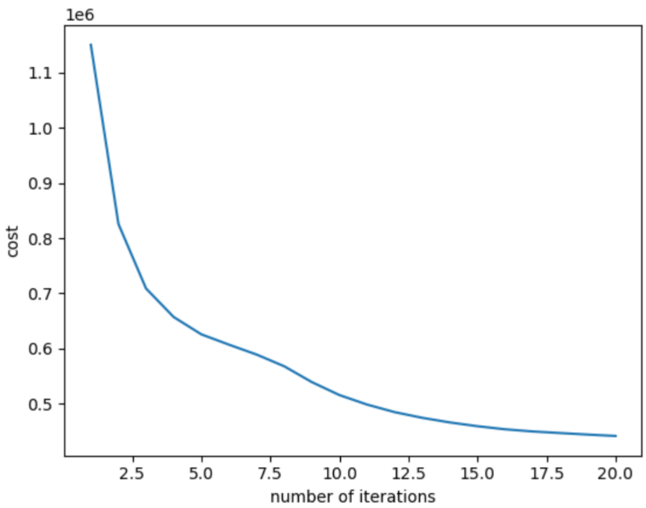
The cost for c1.txt is:

In [24]:
```
if __name__ == "__main__":
    main()
```



The costs for c2.txt is:

In [19]:
```
if __name__ == "__main__":
    main()
```



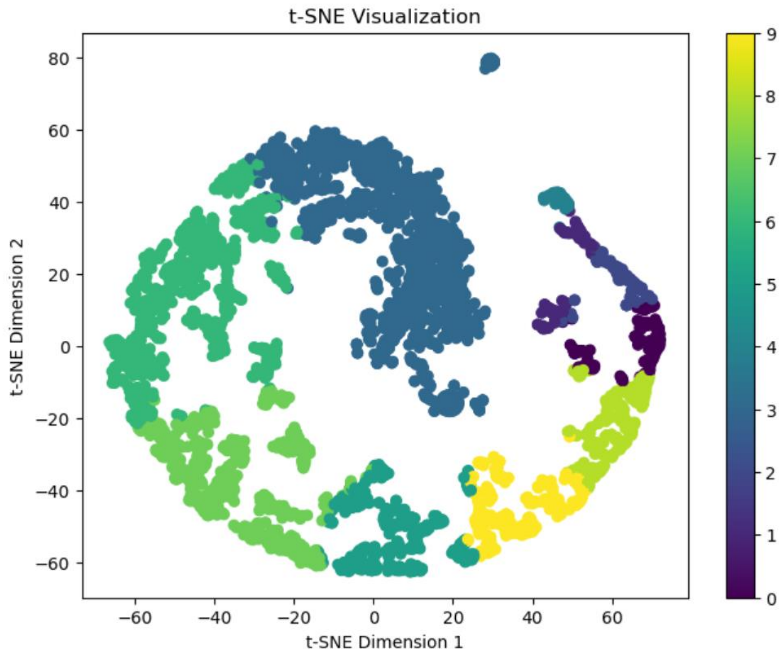
(3) T-SNE

The modified code is:

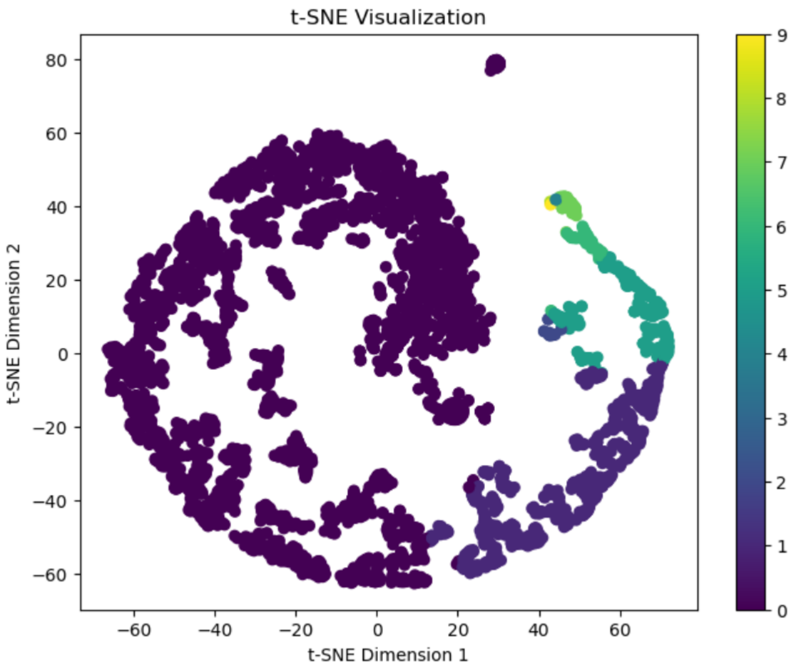
```
    # iterative k-means
    costs=[]
    for _ in range(MAX_ITER):
        # Transform each point to a combo of point, closest centroid, count=1
        # point -> (closest_centroid, (point, 1))
        combo = data.map(lambda point: (closest(point, centroids, norm), (point, 1)))

        # Re-compute cluster center
        # For each cluster center (key), aggregate its values
        # by summing up points and count
        aggregated_combo = combo.reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))

        # Average the points for each centroid: divide sum of points by count
        updated_centroids = aggregated_combo.mapValues(lambda value: value[0] / value[1])
        # Use collect() to turn RDD into list
        centroids = updated_centroids.map(lambda x: x[1]).collect()

        cost = combo.map(lambda x: linalg.norm(x[1][0] - centroids[x[0]], norm)).sum()
        costs.append(cost)


    out = combo.map(lambda x: (x[0],x[1][0]))

    return costs, centroids, out
```

```
def main():
    # Spark settings
    sc = SparkContext.getOrCreate()

    # Load the data, cache this since we're accessing this each iteration
    data = sc.textFile(DATA_PATH).map(
            lambda line: np.array([float(x) for x in line.split()])
            ).cache()

    # Load the initial centroids c1, split into a list of np arrays
    centroids1 = sc.textFile(C1_PATH).map(
            lambda line: np.array([float(x) for x in line.split(' ')])
            ).collect()

    # Load the initial centroids c2, split into a list of np arrays
    centroids2 = sc.textFile(C2_PATH).map(
            lambda line: np.array([float(x) for x in line.split(' ')])
            ).collect()

    cost1, centroid1, out = kmeans(data, centroids1, norm=2)
    points = out.map(lambda x: x[1]).collect()
    points = np.array(points)
    tsne = TSNE(n_components=2, random_state=42)
    low_dimensional_points = tsne.fit_transform(points)
    clusters = out.map(lambda x: x[0]).collect()

    plt.figure(figsize=(8, 6))
    plt.scatter(low_dimensional_points[:, 0], low_dimensional_points[:, 1], c=clusters, cmap='viridis')
    plt.title('t-SNE Visualization')
    plt.xlabel('t-SNE Dimension 1')
    plt.ylabel('t-SNE Dimension 2')
    plt.colorbar()
    plt.show()
```

The output figure of “c1” is:



The output figure of “c2” is:



(4) Cost comparison

For L1 distance, the cost of c1.txt is smaller than c2.txt, which means c1.txt is better.

For L2 distance, the cost of c1.txt is also smaller than c2.txt.

(5) Time Complexity

The time complexity of the k-means algorithm is O(n). The time spent on this algorithm linearly depends on the number of input points. For every iteration, the program must check all the points to calculate the new centroids.

2. Binary Classification on Spark

(1) The code for importing data is:

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
import pandas as pd
import numpy as np
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator
from pyspark.mllib.evaluation import BinaryClassificationMetrics
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.classification import NaiveBayes
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.classification import GBTClassifier
from pyspark.ml.classification import MultilayerPerceptronClassifier
from pyspark.ml.classification import LinearSVC
from pyspark.ml.classification import OneVsRest
```

```
class CurveMetrics(BinaryClassificationMetrics):
    def __init__(self, *args):
        super(CurveMetrics, self).__init__(*args)

    def _to_list(self, rdd):
        points = []
        # Note this collect could be inefficient for large datasets
        # considering there may be one probability per datapoint (at most)
        # The Scala version takes a numBins parameter,
        # but it doesn't seem possible to pass this from Python to Java
        for row in rdd.collect():
            # Results are returned as type scala.Tuple2,
            # which doesn't appear to have a py4j mapping
            points += [(float(row._1()), float(row._2()))]
        return points

    def get_curve(self, method):
        rdd = getattr(self._java_model, method)().toJavaRDD()
        return self._to_list(rdd)
```

```
sc = SparkSession \
    .builder \
    .appName("binary") \
    .getOrCreate()
```

```
df = sc.read.csv("gs://6893_bucket_1/HW2/adult.data.csv", inferSchema=True)
df = df.withColumnRenamed("_c0", "age") \
    .withColumnRenamed("_c1", "workclass") \
    .withColumnRenamed("_c2", "fnlwgt") \
    .withColumnRenamed("_c3", "education") \
    .withColumnRenamed("_c4", "education_num") \
    .withColumnRenamed("_c5", "marital_status") \
    .withColumnRenamed("_c6", "occupation") \
    .withColumnRenamed("_c7", "relationship") \
    .withColumnRenamed("_c8", "race") \
    .withColumnRenamed("_c9", "sex") \
    .withColumnRenamed("_c10", "capital_gain") \
    .withColumnRenamed("_c11", "capital_loss") \
    .withColumnRenamed("_c12", "hours_per_week") \
    .withColumnRenamed("_c13", "native_country") \
    .withColumnRenamed("_c14", "income")
```

(2) The code for data preprocessing is:

```
categorical_cols = ["workclass", "education", "marital_status", "occupation", "relationship", "race", "sex", "native_country", "income"]

indexers = [StringIndexer(inputCol=col, outputCol=col + "_index", handleInvalid="keep") for col in categorical_cols]
encoder = OneHotEncoder(inputCols=[indexer.getOutputCol() for indexer in indexers], outputCols=[col + "_encoded" for col in categorical_cols])
```

```
feature_cols = ["age", "fnlwgt", "education_num", "capital_gain", "capital_loss", "hours_per_week"] + [col + "_encoded" for col in categorical
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
```

```
from pyspark.sql.functions import when
pipeline = Pipeline(stages=indexers + [encoder, assembler])
# Fit and transform the data using the pipeline
preprocessed_data = pipeline.fit(df).transform(df)
preprocessed_data = preprocessed_data.withColumn("income_index", when(preprocessed_data["income_index"] == 1.0, 1.0).otherwise(0.0))
train_data, test_data = preprocessed_data.randomSplit([0.7, 0.3], seed=100)
evaluator = MulticlassClassificationEvaluator(labelCol="income_index", predictionCol="prediction")
```

(3) The code for training models and check accuracy is:

In [ ]:
```
#Logistic Regression
lr = LogisticRegression(labelCol="income_index", featuresCol="features")
# Train the model
lr_model = lr.fit(train_data)
predictions = lr_model.transform(test_data)
```

In [ ]:
```
print("Logistic Regression")
accuracy = evaluator.evaluate(predictions, {evaluator.metricName:"accuracy"})
print(f"accuracy: {accuracy}")
```

In [ ]:
```
#random forest

rf = RandomForestClassifier(labelCol="income_index", featuresCol="features", numTrees=100)
rf_model = rf.fit(train_data)
predictions_rf = rf_model.transform(test_data)
```

In [ ]:
```
accuracy = evaluator.evaluate(predictions_rf, {evaluator.metricName:"accuracy"})
print(f"accuracy: {accuracy}")
```

In [ ]:
```
#naive bayes
# create the trainer and set its parameters
nb = NaiveBayes(smoothing=1.0, modelType="multinomial",labelCol="income_index", featuresCol="features")

# train the model
nb_model = nb.fit(train_data)

# select example rows to display.
predictions_nb = nb_model.transform(test_data)
```

In [ ]:
```
accuracy = evaluator.evaluate(predictions_nb, {evaluator.metricName:"accuracy"})
print(f"accuracy: {accuracy}")
```

In [ ]:
```
# Train a DecisionTree model.
dt = DecisionTreeClassifier(featuresCol="features", labelCol="income_index")

# Train model.  This also runs the indexer.
model_dt = dt.fit(train_data)

# Make predictions.
predictions_dt = model_dt.transform(test_data)
```

In [ ]:
```
accuracy = evaluator.evaluate(predictions_dt, {evaluator.metricName:"accuracy"})
print(f"accuracy: {accuracy}")
```

In [ ]:
```
# Train a GBT model.
gbt = GBTClassifier(labelCol="income_index", featuresCol="features", maxIter=10)

# Train model.  This also runs the indexers.
model_gbt = gbt.fit(train_data)

# Make predictions.
predictions_gbt = model_gbt.transform(test_data)
```

In [ ]:
```
accuracy = evaluator.evaluate(predictions_gbt, {evaluator.metricName:"accuracy"})
print(f"accuracy: {accuracy}")
```

In [ ]:
```
# create the trainer and set its parameters
layers = [108, 5, 4, 2]
trainer = MultilayerPerceptronClassifier(maxIter=100, layers=layers, seed=1234, labelCol="income_index", featuresCol="features",solver="gd", s

# train the model
model_mpc = trainer.fit(train_data)

# compute accuracy on the test set
predictions_mpc = model_mpc.transform(test_data)
```

In [ ]:
```
accuracy = evaluator.evaluate(predictions_mpc, {evaluator.metricName:"accuracy"})
print(f"accuracy: {accuracy}")
```

In [ ]:
```
lsvc = LinearSVC(maxIter=10, regParam=0.1,labelCol="income_index", featuresCol="features")

# Fit the model
model_lsvc = lsvc.fit(train_data)
predictions_lsvc = model_lsvc.transform(test_data)
```

In [ ]:
```
accuracy = evaluator.evaluate(predictions_lsvc, {evaluator.metricName:"accuracy"})
print(f"accuracy: {accuracy}")
```

```
In [ ]: #one over rest
lr = LogisticRegression(maxIter=10, tol=1e-6, fitIntercept=True)


ovr = OneVsRest(classifier=lr, labelCol="income_index", featuresCol="features")


# 训练模型
model_ovr = ovr.fit(train_data)

# 进行预测
predictions_ovr = model_ovr.transform(test_data)
```

```
In [ ]: accuracy = evaluator.evaluate(predictions_ovr, {evaluator.metricName:"accuracy"})
print(f"accuracy: {accuracy}")
```

(4) Accuracies and comparations

The accuracy of logistic regression method is:

```
In [11]: print("Logistic Regression")
accuracy = evaluator.evaluate(predictions, {evaluator.metricName:"accuracy"})
print(f"accuracy: {accuracy}")

Logistic Regression
[Stage 136:>                                                    (0 + 1) / 1]
accuracy: 0.8496248329735842
```

The accuracy of random forest method is:

```
In [9]: #random forest

rf = RandomForestClassifier(labelCol="income_index", featuresCol="features", numTrees=100)
rf_model = rf.fit(train_data)
predictions_rf = rf_model.transform(test_data)
```

```
In [10]: accuracy = evaluator.evaluate(predictions_rf)
print(f"accuracy: {accuracy}")

[Stage 43:>                                                    (0 + 1) / 1]
accuracy: 0.831534587316271
```

The accuracy of Naïve Bayes is:

```
In [20]: accuracy = evaluator.evaluate(predictions_nb, {evaluator.metricName:"accuracy"})
print(f"accuracy: {accuracy}")

[Stage 177:>                                                    (0 + 1) / 1]
accuracy: 0.7829170521122417
```

The accuracy of Decision Tree Regression is:

```
accuracy = evaluator.evaluate(predictions_dt, {evaluator.metricName:"accuracy"})
print(f"accuracy: {accuracy}")

accuracy: 0.8383184294377634
```

The accuracy of Gradient-boosted tree classifier is:

```
In [32]: accuracy = evaluator.evaluate(predictions_gbt, {evaluator.metricName:"accuracy"})
         print(f"accuracy: {accuracy}")

accuracy: 0.8471579812930414
```

The accuracy of Multilayer Perceptron Classifier is:

```
In [33]: accuracy = evaluator.evaluate(predictions_mpc, {evaluator.metricName:"accuracy"})
         print(f"accuracy: {accuracy}")

accuracy: 0.7567067530064755
```

The accuracy of Linear Support Vector Machine is:

```
In [36]: accuracy = evaluator.evaluate(predictions_lsvc, {evaluator.metricName:"accuracy"})
         print(f"accuracy: {accuracy}")

[Stage 855:>                                                  (0 + 1) / 1]

accuracy: 0.8372905745708706
```

The accuracy of one-over-rest is:

```
accuracy = evaluator.evaluate(predictions_ovr, {evaluator.metricName:"accuracy"})
print(f"accuracy: {accuracy}")

[Stage 884:>                                                  (0 + 1) / 1]

accuracy: 0.8490081200534485
```

Overall, the logistic regression method has the highest accuracy.
The accuracy sequence is:
Logistic regression > One-Over-rest > Gradient-boosted tree classifier > Decision Tree Regression > Linear Support Vector Machine > Random Forest > Naïve Bayes > Multilayer Perceptron Classifier

3. Monitoring Hadoop metrics

(1) Verify Hadoop installation

```
2023-10-06 04:52:17,607 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = instance-1/10.142.0.9
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 3.3.4
STARTUP_MSG:   classpath = /usr/local/hadoop/etc/hadoop:/usr/loc
```

```
2023-10-06 04:52:19,892 INFO namenode.FSImageFormatProtobuf: Image file /home/hadoop/hadoopinfra/hdfs/name
ckpt_0000000000000000000 of size 401 bytes saved in 0 seconds .
2023-10-06 04:52:19,927 INFO namenode.NNStorageRetentionManager: Going to retain 1 images with txid >= 0
2023-10-06 04:52:19,969 INFO namenode.FSNamesystem: Stopping services started for active state
2023-10-06 04:52:19,970 INFO namenode.FSNamesystem: Stopping services started for standby state
2023-10-06 04:52:19,978 INFO namenode.FSImage: FSImageSaver clean checkpoint: txid=0 when meet shutdown.
2023-10-06 04:52:19,983 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at instance-1/10.142.0.9
************************************************************/
```

```
hadoop@instance-1:~$ start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [instance-1]
```

```
hadoop@instance-1:~$ start-yarn.sh
Starting resourcemanager
Starting nodemanagers
```

(2) HDFS metrics monitoring

Execute the “wordcount” program.

Monitor the metrics via HDFS NameNode:

Security is off.

Safemode is off.

38 files and directories, 4 blocks (4 replicated blocks, 0 erasure coded block groups) = 42 total filesystem object(s).

Heap Memory used 143.05 MB of 244.47 MB Heap Memory. Max Heap Memory is 3.09 GB.

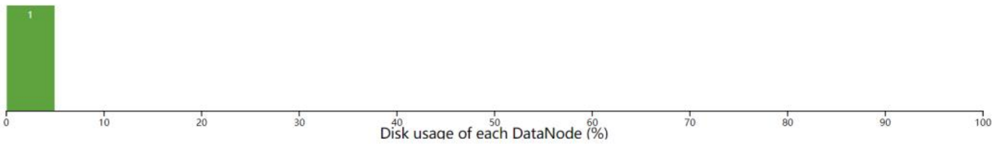
Non Heap Memory used 98.4 MB of 101.42 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	491.9 GB
Configured Remote Capacity:	0 B
DFS Used:	218.86 KB (0%)
Non DFS Used:	14.67 GB
DFS Remaining:	457.12 GB (92.93%)
Block Pool Used:	218.86 KB (0%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	1 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion (including replicas)	0
Block Deletion Start Time	Fri Oct 06 23:08:45 +0800 2023
Last Checkpoint Time	Fri Oct 06 23:08:24 +0800 2023
Enabled Erasure Coding Policies	RS-6-3-1024k

Datanode Information

✔ In service | Down | Decommissioning | Decommissioned | Decommissioned & dead
Entering Maintenance | In Maintenance | In Maintenance & dead

Datanode usage histogram



In operation

DataNode State All   Show 25 entries   Search:

Node	Http Address	Last contact	Last Block Report	Used	Non DFS Used	Capacity	Blocks	Block pool used	Version
✔/default-rack/cluster-hadoop-m.c.eecs6893-399118.internal:9866 (10.128.0.6:9866)	http://cluster-hadoop-m.c.eecs6893-399118.internal:9864	1s	28m	218.86 KB	14.67 GB	491.9 GB	4	218.86 KB (0%)	3.3.6

Showing 1 to 1 of 1 entries   Previous 1 Next

Part of the metrics:


```
{
  "beans" : [ {
    "name" : "Hadoop:service=NameNode,name=DelegationTokenSecretManagerMetrics",
    "modelerType" : "DelegationTokenSecretManagerMetrics",
    "tag.Context" : "token",
    "tag.Hostname" : "cluster-hadoop-m",
    "RemoveTokenNumOps" : 0,
    "RemoveTokenAvgTime" : 0.0,
    "StoreTokenNumOps" : 0,
    "StoreTokenAvgTime" : 0.0,
    "TokenFailure" : 0,
    "UpdateTokenNumOps" : 0,
    "UpdateTokenAvgTime" : 0.0
  }, {
    "name" : "Hadoop:service=NameNode,name=JvmMetrics",
    "modelerType" : "JvmMetrics",
    "tag.Context" : "jvm",
    "tag.ProcessName" : "NameNode",
    "tag.SessionId" : null,
    "tag.Hostname" : "cluster-hadoop-m",
    "MemNonHeapUsedM" : 98.83336,
    "MemNonHeapCommittedM" : 101.79297,
    "MemNonHeapMaxM" : -1.0,
    "MemHeapUsedM" : 145.86484,
    "MemHeapCommittedM" : 244.46875,
    "MemHeapMaxM" : 3168.75,
    "MemMaxM" : 3168.75,
    "GcCountParNew" : 12,
    "GcTimeMillisParNew" : 473,
    "GcCountConcurrentMarkSweep" : 3,
    "GcTimeMillisConcurrentMarkSweep" : 98,
    "GcCount" : 15,
    "GcTimeMillis" : 571,
    "GcNumWarnThresholdExceeded" : 0,
    "GcNumInfoThresholdExceeded" : 0,
    "GcTotalExtraSleepTime" : 45,
    "GcTimePercentage" : 0,
    "ThreadsNew" : 0,
    "ThreadsRunnable" : 13,
    "ThreadsBlocked" : 0,
    "ThreadsWaiting" : 5,
    "ThreadsTimedWaiting" : 62,
    "ThreadsTerminated" : 0,
    "LogFatal" : 0,
    "LogError" : 0,
    "LogWarn" : 4,
    "LogInfo" : 218
  }, {
    "name" : "Hadoop:service=NameNode,name=RpcActivityForPort8051",
    "modelerType" : "RpcActivityForPort8051",
```


Important metrics:

a) "CapacityRemainingGB"

Indicates the remaining storage capacity of HDFS. If the remaining storage capacity is close to zero, it means HDFS will be unable to continue writing data, which may result in data loss or application interruption.

b) "NumLiveDataNodes"

Indicates the number of currently surviving data DataNodes. The number of data nodes directly affects the availability and performance of HDFS. Monitoring this metric can help you ensure that the nodes in your cluster are functioning properly.

c) “BlocksTotal”

Represents the total number of data blocks in HDFS. This can help you understand the size of the data in the cluster and the storage situation of HDFS.

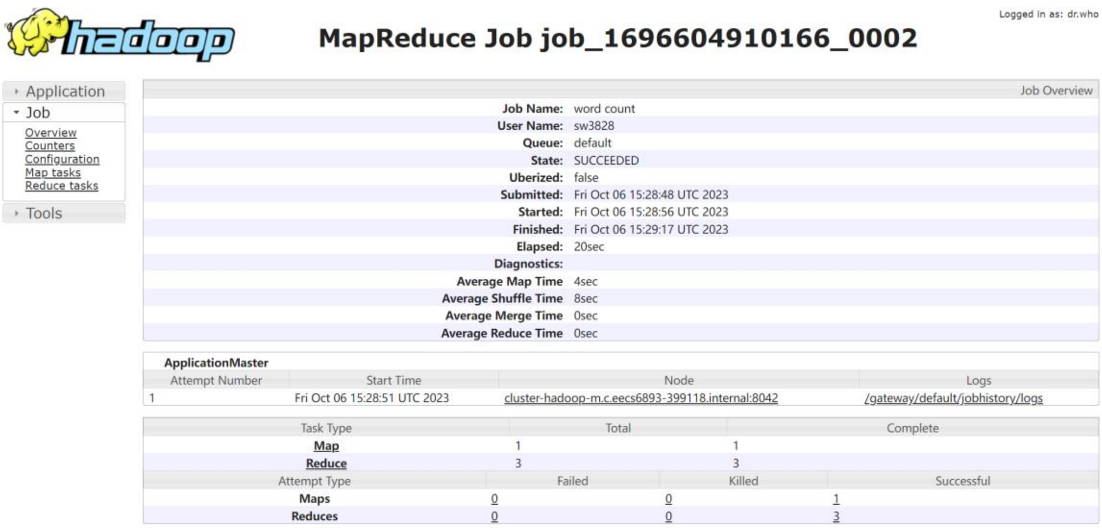
d) “VolumeFailuresTotal”

Indicates the number of volume failures across all DataNodes. Volume failures can lead to data unavailability. Monitoring this metric helps in identifying hardware issues such as disk failures, enabling timely replacements or repairs to maintain data integrity and availability.

e) “UnderReplicatedBlocks”

It provides the number of blocks that have a replication level less than the specified level. Monitoring under-replicated blocks is critical for data durability and fault tolerance. High numbers indicate a potential data loss risk.

(3) MapReduce counters monitoring



	Name	Map	Reduce	Total
File System Counters	FILE: Number of bytes read	0	84,637	84,637
	FILE: Number of bytes written	372,090	946,687	1,318,777
	FILE: Number of large read operations	0	0	0
	FILE: Number of read operations	0	0	0
	FILE: Number of write operations	0	0	0
	HDFS: Number of bytes read	120,089	0	120,089
	HDFS: Number of bytes read erasure-coded	0	0	0
	HDFS: Number of bytes written	0	59,783	59,783
	HDFS: Number of large read operations	0	0	0
	HDFS: Number of read operations	3	15	18
	HDFS: Number of write operations	0	9	9

	Name	Map	Reduce	Total
Job Counters	Data-local map tasks	0	0	1
	Killed reduce tasks	0	0	1
	Launched map tasks	0	0	1
	Launched reduce tasks	0	0	3
	Total megabyte-milliseconds taken by all map tasks	0	0	14,705,398
	Total megabyte-milliseconds taken by all reduce tasks	0	0	91,547,282
	Total time spent by all map tasks (ms)	0	0	4,343
	Total time spent by all maps in occupied slots (ms)	0	0	14,705,398
	Total time spent by all reduce tasks (ms)	0	0	27,037
	Total time spent by all reduces in occupied slots (ms)	0	0	91,547,282
	Total vcore-milliseconds taken by all map tasks	0	0	4,343
	Total vcore-milliseconds taken by all reduce tasks	0	0	27,037

	Name	Map	Reduce	Total
Map-Reduce Framework	Combine input records	20,366	0	20,366
	Combine output records	6,282	0	6,282
	CPU time spent (ms)	1,130	3,600	4,730
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	23	69	92
	Input split bytes	104	0	104
	Map input records	3,667	0	3,667
	Map output bytes	194,984	0	194,984
	Map output materialized bytes	84,637	0	84,637
	Map output records	20,366	0	20,366
	Merged Map outputs	0	3	3
	Peak Map Physical memory (bytes)	595,791,872	0	595,791,872
	Peak Map Virtual memory (bytes)	4,825,849,856	0	4,825,849,856
	Peak Reduce Physical memory (bytes)	0	421,457,920	421,457,920
	Peak Reduce Virtual memory (bytes)	0	4,831,727,616	4,831,727,616
	Physical memory (bytes) snapshot	595,791,872	1,236,316,160	1,832,108,032
	Reduce input groups	0	6,282	6,282
	Reduce input records	0	6,282	6,282
	Reduce output records	0	6,282	6,282
	Reduce shuffle bytes	0	84,637	84,637
	Shuffled Maps	0	3	3
	Spilled Records	6,282	6,282	12,564
	Total committed heap usage (bytes)	587,202,560	1,048,576,000	1,635,778,560
	Virtual memory (bytes) snapshot	4,825,849,856	14,485,041,152	19,310,891,008

	Name	Map	Reduce	Total
Shuffle Errors	BAD_ID	0	0	0
	CONNECTION	0	0	0
	IO_ERROR	0	0	0
	WRONG_LENGTH	0	0	0
	WRONG_MAP	0	0	0
	WRONG_REDUCE	0	0	0

	Name	Map	Reduce	Total
File Input Format Counters	Bytes Read	119,985	0	119,985

	Name	Map	Reduce	Total
File Output Format Counters	Bytes Written	0	59,783	59,783

File-system counters record file system operations during a job, including the number of read and write operations and the total number of bytes read and written. These counters help developers understand the access pattern, frequency and amount of data to HDFS by jobs.

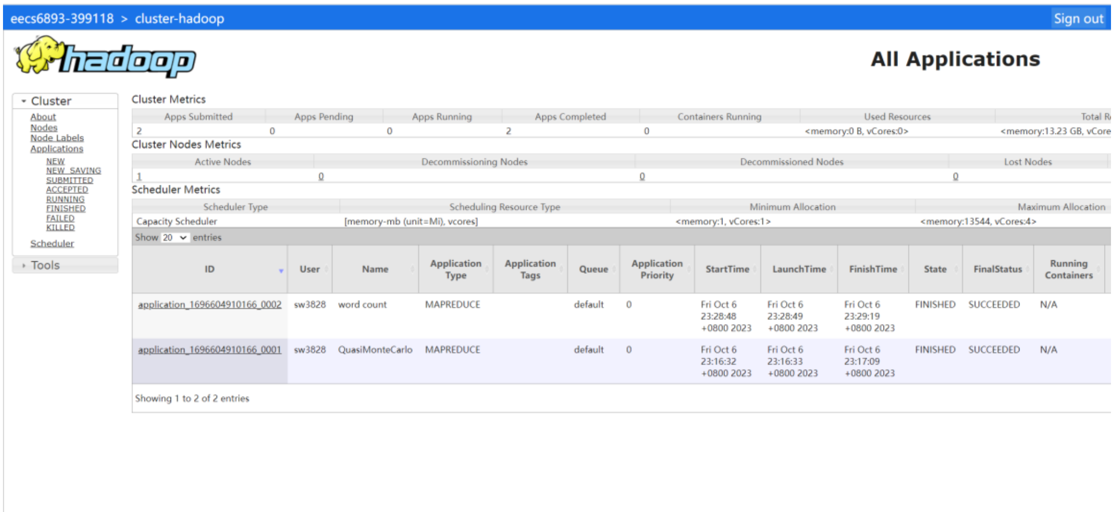
Job Counters provide overall statistical information about jobs, including the number of tasks, start time, completion time, number of failed tasks, etc., helping developers understand the progress and execution of jobs.

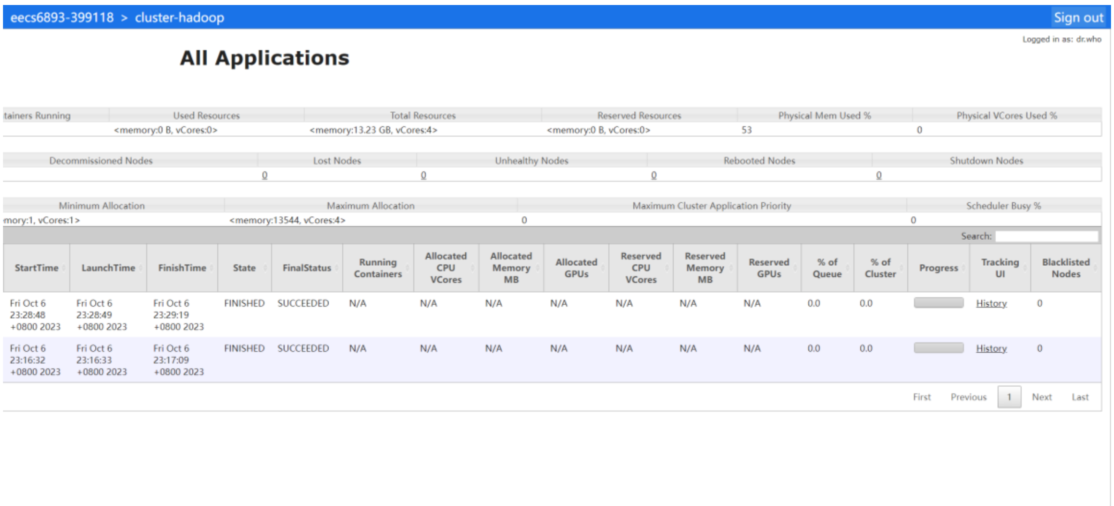
Map-Reduce Framework Provides information about the number of calls to the Mapper and Reducer functions, the number of input and output records, etc. during the execution of a MapReduce job. These counters help developers understand the execution of their jobs, including detailed statistics on data processing and transformations.

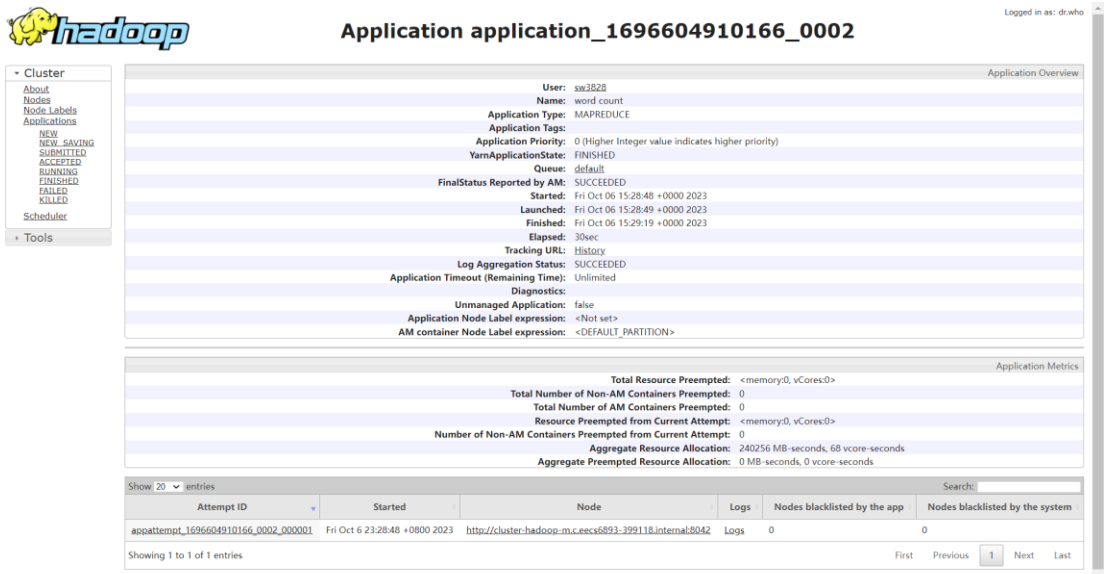
Shuffle Errors provides the number of errors, error types, failed task information, etc. By analyzing the Shuffle Errors counter, we can identify problems that occur during the Shuffle phase of job execution and take corresponding measures to solve these problems, thereby improving job performance and stability.

File Input Format Counters and File Output Format Counters record the size of the input file and output file.

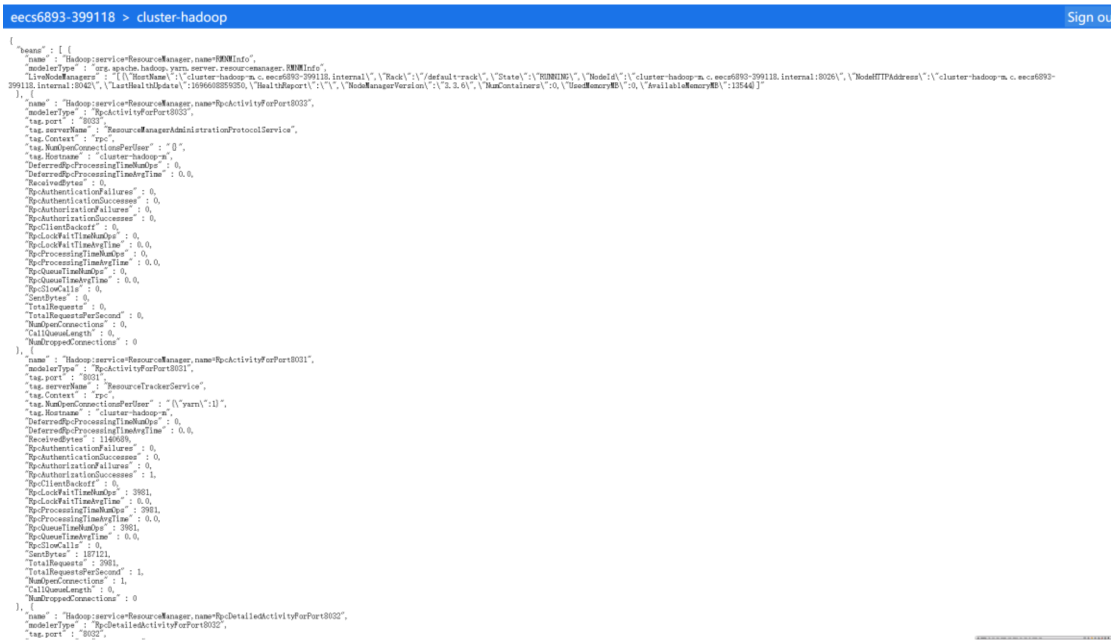
(4) YARN metrics monitoring







Part of the metrics:



Important metrics:

(a) "AvailableMB"

This metric represents the amount of memory currently available in the YARN cluster. Understanding available memory can help you determine whether there are enough resources to start a new application or container.

(b) "AllocatedMB"

This metric represents the amount of memory that has been allocated in the YARN cluster. Understanding allocated memory can help you determine the current load on your cluster and whether resource adjustments are needed.

(c) "AppsSubmitted"

This metric represents the total number of applications currently running in the YARN cluster. Knowing the number of applications helps you understand the load and resource requirements of your cluster.

(d) “AggregateContainersAllocated”
This metric provides information on cluster resource utilization. By monitoring the number of allocated containers, you can understand the amount of resources currently being used in the cluster, helping you better manage resources, improve cluster utilization, and ensure that applications are getting enough resources to perform.
(e) “containersRunning”
This metric represents the number of containers currently running in the YARN cluster. A running container refers to a container that has been allocated resources and is executing tasks. Monitoring this metric can help you understand the current workload and resource utilization in your cluster. By tracking the number of running containers, you can ensure that cluster resources are being used effectively, and it can also help you monitor the performance and stability of your cluster.