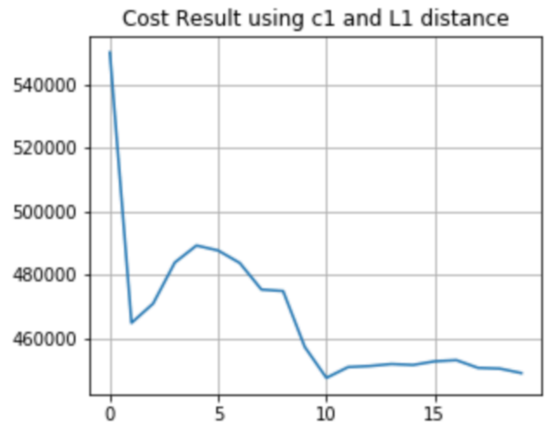
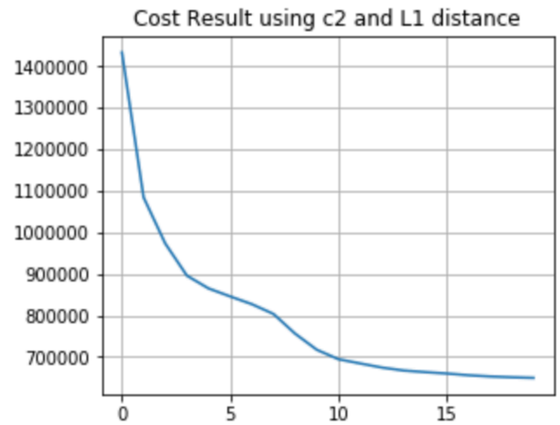
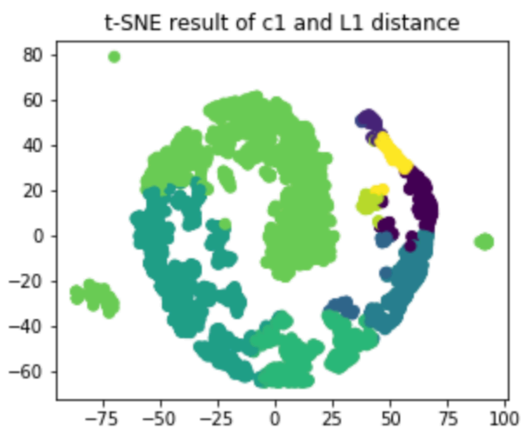
Homework1

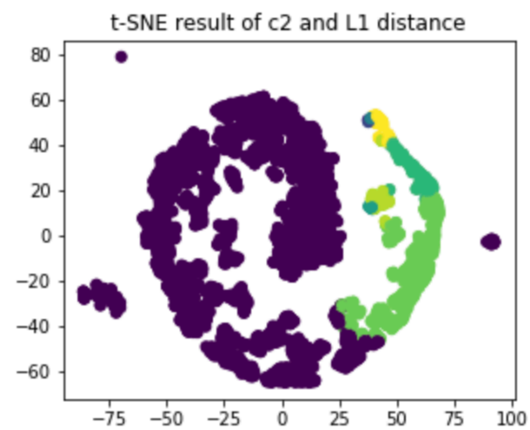
1. Iterative K-means clustering on Spark

(1)

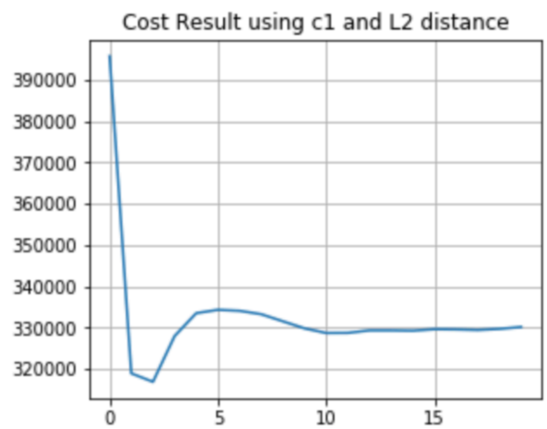


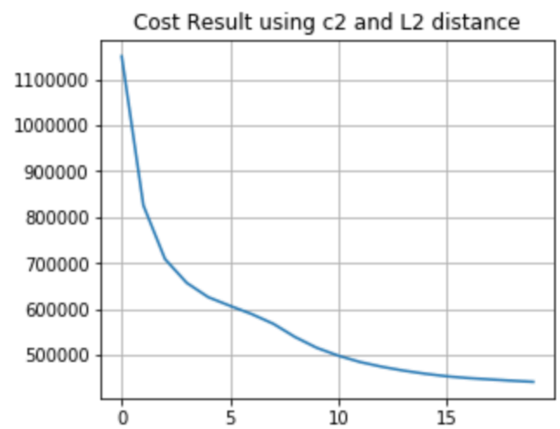


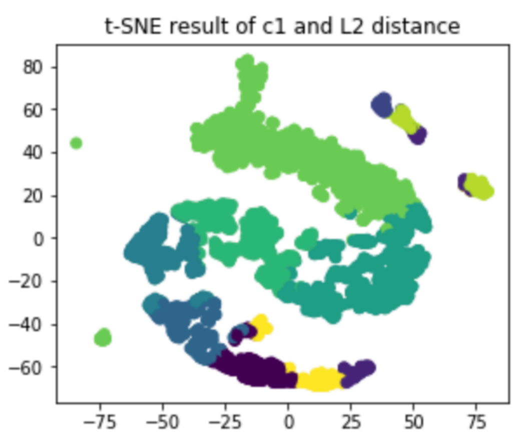


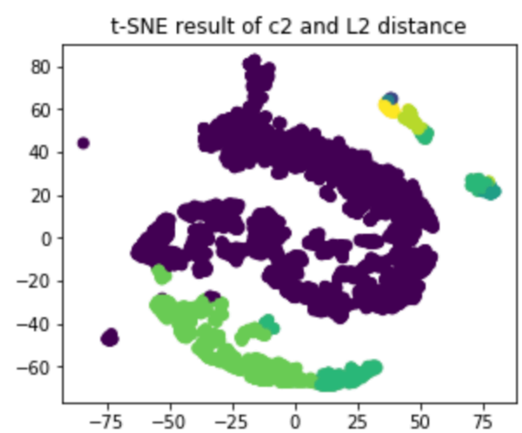


(2) and (3)









(4)

No, although from the above plots we could see that using c1 will get a better cost than c2 in the end, but we only got 20 iterations. If the we increase the 'MAX_ITER', the cost by using c2 will eventually be lower than by using c1.

Because random initialization sometimes can results in creating centroids in such a way that they are clumped together in space. Then in the end we'll have several clusters cramped tightly, and make other clusters more sparse, which will result in a higher cost.

(5)

Assume we have:

`k` clusters

`p` data points, with `d` dimensions

`n` maximum iterations

Then:

for calculating the distance between each points and centroids: O(n) = pdk

for choosing the closest centroids by using quick sort: O(n) = p * klog(k)

for recomputing the controids: O(n) = pd

So altogether, O(n) = n(pdk + pklog(k) + pd) = np(dk + klog(k) + d)

Code:

```
import operator
import sys
from pyspark import SparkConf, SparkContext
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg # calculating L1 and L2 distance
import time # calculate computing time
import pandas as pd
from sklearn.manifold import TSNE # for dimensionality reduction
import random

# Macros.
MAX_ITER = 20
DATA_PATH = "gs://big_data_hw/hw1/data.txt"
C1_PATH = "gs://big_data_hw/hw1/c1.txt"
C2_PATH = "gs://big_data_hw/hw1/c2.txt"

# Helper functions.
def closest(p, centroids, norm):
    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=2):
    cost = []
                            References:
    for t in range(MAX_ITER) temp.py:20
        combo = data.map(lam temp.py:33 losest(point, centroids, norm), (point, 1))).cache()

        #within cluser cost and running time
        costById = combo.map(lambda x: (x[0], linalg.norm(x[1][0] - centroids[x[0]], norm))) \
        .reduceByKey(lambda x, y: x + y).collect()
        cost.append(sum([costById[i][1] for i in range(len(costById))]))

        reduce1 = combo.reduceByKey(lambda a, b: ([a[0][i]+b[0][i] for i in range(len(a[0]))], a[1]+b[1]))

        # Average the points for each centroid: divide sum of points by count
        map1 = reduce1.sortByKey().map(lambda x: [x[1][0][i] / x[1][1] for i in range(len(x[1][0]))])

        # Use collect() to turn RDD into list
        centroids = map1.collect()

    centroIndex = data.map(lambda point: closest(point, centroids, norm)).collect()

    return cost, centroIndex

def main(norm):
    random.seed(0)
    # Spark settings
    conf = SparkConf()
    sc = SparkContext.getOrCreate(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()

    # calculate the cost and the final central point index
    cost1, centroIndex1 = kmeans(data, centroids1, norm)
    cost2, centroIndex2 = kmeans(data, centroids2, norm)

    # demensionality reduction
    data_np = np.array(data.collect())
    data_embedded = TSNE(n_c        it_transform(data_np)
    vis_x = data_embedded[:,  References:
    vis_y = data_embedded[:,  temp.py:34
                              temp.py:43
    plt.figure(figsize=(10,8  temp.py:45
    # plot for cost           temp.py:62
    plt.subplot(2,2,1)        temp.py:66
    plt.plot(range(20), cost_
    plt.title('Cost Result using c1 and L%d distance' % (norm))
    plt.grid(True)
    plt.subplot(2,2,2)
    plt.plot(range(20), cost2)
    plt.title('Cost Result using c2 and L%d distance' % (norm))
    plt.grid(True)

    # plot for clustering result
    plt.subplot(2,2,3)
    plt.scatter(vis_x, vis_y, c = centroIndex1)
    plt.title('t-SNE result of c1 and L%d distance' % (norm))
    plt.subplot(2,2,4)
    plt.scatter(vis_x, vis_y, c = centroIndex2)
    plt.title('t-SNE result of c2 and L%d distance' % (norm))

    plt.subplots_adjust(top=0.92, bottom=0.08, left=0.10,
                        right=0.95, hspace=0.35, wspace=0.35)

    plt.show()

    sc.stop()
```

2. Binary classification with Spark MLlib

(1) data loading

```
In [1]: # create SparkSession and load data
        from pyspark.sql import SparkSession
        ADULT_PATH = "gs://big_data_hw/hw1/adult.data.csv"

        spark = SparkSession \
            .builder \
            .appName("Python Spark SQL basic example") \
            .config("spark.some.config.option", "some-value") \
            .getOrCreate()

        df = spark.read.csv(ADULT_PATH, inferSchema = True)
        # rename each column
        oldColumns = df.columns
        newColumns = ("age", "workclass", "fnlwgt", "education",
                      "education_num", "marital_status", "occupation",
                      "relationship", "race", "sex", "capital_gain",
                      "capital_loss", "hours_per_week", "native_country", "income")

        df = reduce(lambda data, idx: data.withColumnRenamed(oldColumns[idx], newColumns[idx]), xrange(len(oldColumns)), df)
```

(2) preprocessing

In [35]:
```
# preprocessing and model building
from pyspark.ml import Pipeline
from pyspark.ml.feature import *
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator
from pyspark.mllib.evaluation import MulticlassMetrics
import matplotlib.pyplot as plt
```

In [5]:
```
# piping the preprocess and regression
categorical_variables = ['workclass', 'education', 'marital_status',
                         'occupation', 'relationship', 'race', 'sex', 'native_country']
continuous_variables = ['age', 'fnlwgt', 'education_num', 'capital_gain', 'capital_loss', 'hours_per_week']

indexers = [StringIndexer(inputCol=column,
                          outputCol=column+"_index") for column in categorical_variables]
encoder = OneHotEncoderEstimator(
    inputCols=[indexer.getOutputCol() for indexer in indexers],
    outputCols=["{0}_encoded".format(indexer.getOutputCol()) for indexer in indexers]
)
assembler = VectorAssembler(
    inputCols=encoder.getOutputCols() + continuous_variables,
    outputCol="features"
)
response = StringIndexer(inputCol='income', outputCol='label')
preprocess = Pipeline(stages=indexers + [encoder, assembler, response])

# split data into 70% training and 30% testing and set the seed to 100
train, test = df.randomSplit([0.7, 0.3], seed = 100)
pre_model = preprocess.fit(train)
train = pre_model.transform(train)
test = pre_model.transform(test)
```

(3) modeling

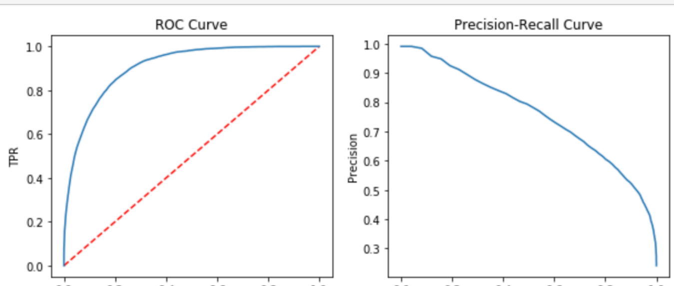
```
lr = LogisticRegression(maxIter=10, featuresCol='features', labelCol='label')
model = lr.fit(train)
prediction = model.transform(test)
```

In [14]:
```
# ROC for training data
plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot([0, 1], [0, 1], 'r--')
plt.plot(model.summary.roc.select('FPR').collect(),
         model.summary.roc.select('TPR').collect())
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('ROC Curve')

plt.subplot(1,2,2)
pr = model.summary.pr.toPandas()
plt.plot(pr['recall'],pr['precision'])
plt.ylabel('Precision')
plt.xlabel('Recall')
plt.title('Precision-Recall Curve')

plt.show()
```



(4) evaluation

In [49]:
```
# testing AUC and accuracy and confusion matrix
evaluator = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction")
evaluator.evaluate(prediction)
print("%s is: %.6f \n" % (evaluator.getMetricName(), evaluator.evaluate(prediction)))

acc = MulticlassClassificationEvaluator(labelCol='label', predictionCol='prediction', metricName='accuracy')
print("%s is: %.6f \n" % (acc.getMetricName(), acc.evaluate(prediction)))

predictionAndLabels = prediction.select('prediction', 'label').rdd
metrics = MulticlassMetrics(predictionAndLabels)
print("confusion matrix is: ")
print(metrics.confusionMatrix())
```

```
areaUnderROC is: 0.902738

accuracy is: 0.848401

confusion matrix is:
DenseMatrix([[6860.,  530.],
             [ 944., 1389.]])
```