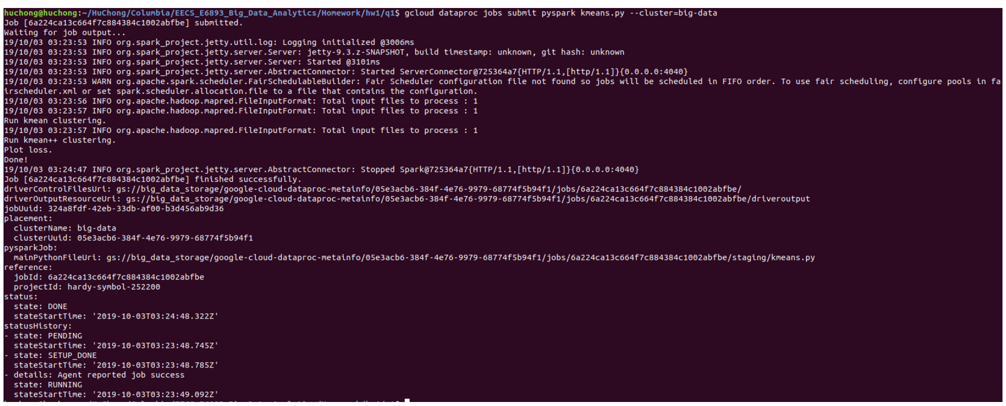
EECS E6893 Big Data Analytic HW1

Chong Hu ch3467

October 3, 2019

Problem 1. `Iterative K-means clustering on Spark`

1. L1 distance
   Here is the terminal command line screenshot.



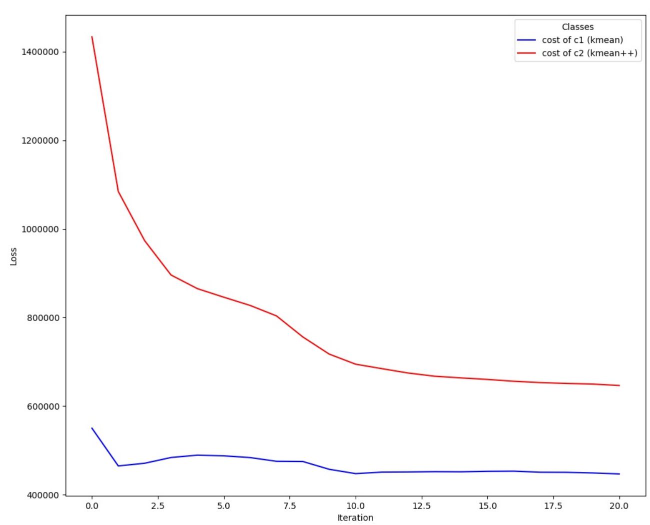
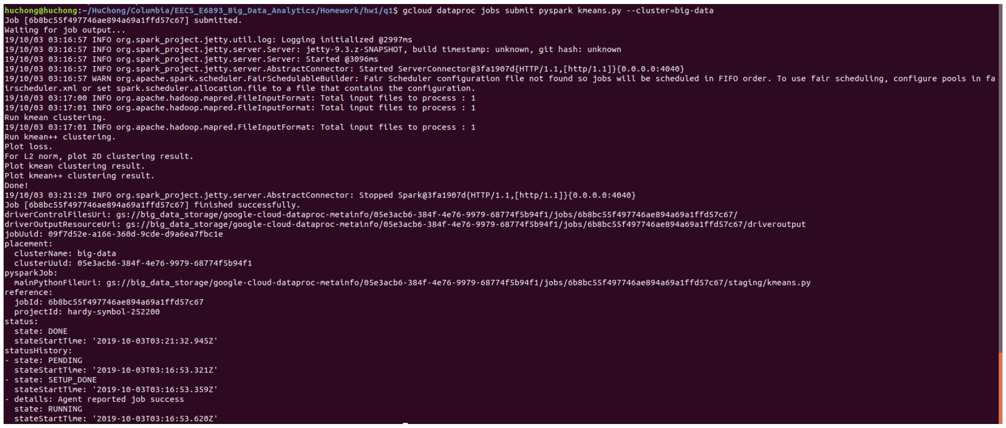


Figure 1: Loss for L1 distance

2. L2 distance
   Here is the terminal command line screenshot.



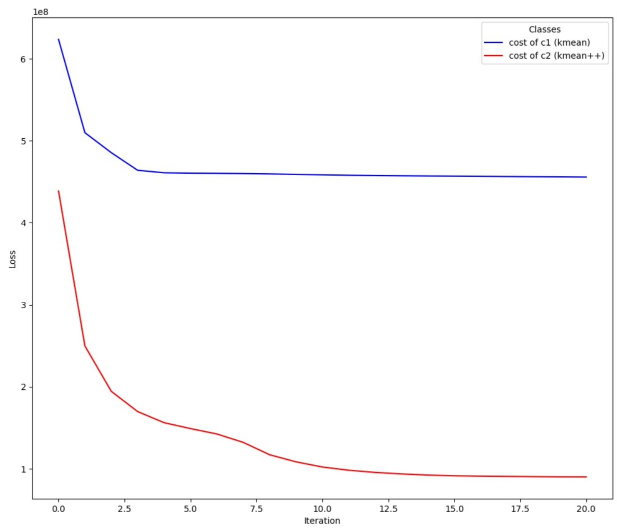


Figure 2: Loss for L2 distance

3. Visualization of high-dimensional data

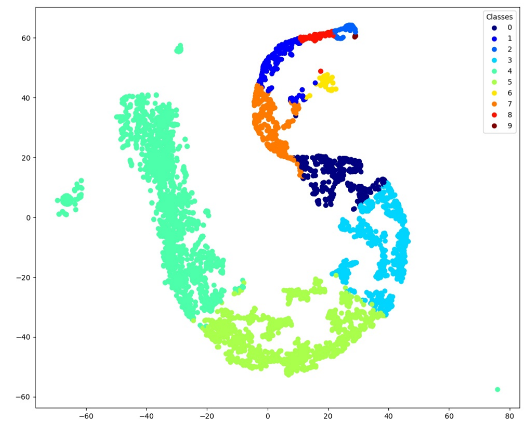


Figure 3: Visualization for kmeans clustering

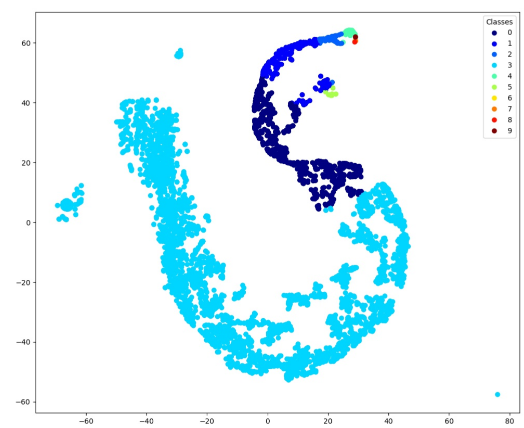


Figure 4: Visualization for kmeans++ clustering

4. For L1 loss, the kmeans++ clustering doesn't provide a better result in loss. But for L2 loss, we can clear see that kmeans++ clustering reduce the loss sharply. In L2, initial cluster centroids which are as far away as possible could provide "good" initial points that make clustering easy to convergence. Those centroids could move to the true centroids as they are re-computed as mean of points in cluster. But in L1, initial cluster centroids which are as far away as possible do not mean they are "good" initial points. Compared with L2 case, they might too far away from most points and hard to move to the true centroids since the edge points has less contribution on the movement. In the loss term, L2 loss play as square term compared with L1.

5. $\mathcal{O}(t*k*n*d)$ where t is number of iterations, k is the number of clusters, n is number of data points and d is number of dimension.

```python
import operator
import sys
from pyspark import SparkConf, SparkContext
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
from sklearn.manifold import TSNE

# Macros.
MAX_ITER = 20
DATA_PATH = "gs://big_data_storage/hw1/data.txt"
C1_PATH = "gs://big_data_storage/hw1/c1.txt"
C2_PATH = "gs://big_data_storage/hw1/c2.txt"
NORM = 1  # change to 2 for l2 loss


# 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) ** norm)
                     for i, c in enumerate(centroids)],
                    key=operator.itemgetter(1))[0]
    return closest_c


def loss(data, centroids, norm=2):
    """

    :param data: original data points in RDD
    :param centroids: centroids used to calculate loss
    :param norm: int 1 or 2
    :return: the loss based on centroids
    """
    norms = data.map(lambda point: linalg.norm(np.subtract(centroids[closest(
        point, centroids, norm=norm)], point), norm) ** norm)
    cost = norms.reduce(lambda norm1, norm2: norm1 + norm2)
```

```python
    return cost


def plot_loss(loss1, loss2, img_path):
    fig = plt.figure(figsize=(12, 10))
    plt.plot(range(len(loss1)), loss1, "b", label="cost of c1 (kmean)")
    plt.plot(range(len(loss2)), loss2, "r", label="cost of c2 (kmean++)")
    plt.legend(loc="upper right", title="Classes")
    plt.xlabel("Iteration")
    plt.ylabel("Loss")
    fig.savefig(img_path)


def plot_cluster(data, img_path):
    index = data.map(lambda x: x[0]).collect()
    points = data.map(lambda x: x[1][0]).collect()
    points_embedded = TSNE(n_components=2, perplexity=50,
                           random_state=100).fit_transform(points)
    fig = plt.figure(figsize=(12, 10))
    scatter = plt.scatter(points_embedded[:, 0], points_embedded[:, 1],
                          marker='o', c=index, cmap='jet')
    plt.legend(*scatter.legend_elements(),
               loc="upper right", title="Classes")
    fig.savefig(img_path)


# K-means clustering
def kmeans(data, centroids, norm=2):
    """
    Conduct k-means clustering given data and centroid.
    This is the basic version of k-means, you might need more
    code to record cluster assignment to plot TSNE, and more
    data structure to record cost.
    Args:
        data (RDD): RDD of points
        centroids (list): A list of centroids points
        norm (int): 1 or 2
    Returns:
        RDD: assignment information of points, a RDD of (centroid, (point, 1))
        centroids: a list of centroids
        loss: a list of loss for each steps.
    """
    # iterative k-means
    # k = len(centroids)

```

```python
    cost_ls = [loss(data=data, centroids=centroids, norm=norm)]

    for _ in range(MAX_ITER):
        # Transform each point to a combo of point, closest centroid, count=1
        # point -> (closest_centroid, (point, 1))

        # Re-compute cluster center
        # For each cluster center (key), aggregate its values
        # by summing up points and count

        # Average the points for each centroid: divide sum of points by count

        # Use collect() to turn RDD into list
        combo = data.map(lambda point: (closest(
            point, centroids, norm=norm), (point, 1)))
        centroids = combo.reduceByKey(lambda combo1, combo2: (
            np.add(combo1[0], combo2[0]),
            combo1[1] + combo2[1])).map(lambda x: np.divide(x[1][0], x[1][1]))
        centroids = centroids.collect()
        cost = loss(data=data, centroids=centroids, norm=norm)
        cost_ls.append(cost)
    combo = data.map(lambda point: (closest(
        point, centroids, norm=norm), (point, 1)))
    return combo, centroids, cost_ls


def main():
    # Spark settings
    conf = SparkConf().setMaster("local").setAppName("kmeans")
    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()
    print("Run kmean clustering.")
    combo1, centroids1, cost1 = kmeans(data=data, centroids=centroids1,
```

```python
                                          norm=NORM)

    print("Run kmean++ clustering.")
    combo2, centroids2, cost2 = kmeans(data=data, centroids=centroids2,
                                          norm=NORM)
    print("Plot loss.")
    plot_loss(cost1, cost2, "loss-l%d.jpg" % NORM)

    if NORM == 2:
        print("For L2 norm, plot 2D clustering result.")
        print("Plot kmean clustering result.")
        plot_cluster(combo1, "kmeans-2Dpoints.jpg")
        print("Plot kmean++ clustering result.")
        plot_cluster(combo2, "kmeans++-2Dpoints.jpg")

    print("Done!")


if __name__ == "__main__":
    main()
```

Problem 2. `Binary classification with Spark MLlib`

For part II, I run the code on my own localhost for convenience.

1. As for the data loading part, please see the python script in detail.

```python
def load(sqlContext, csv_path):
    column_names = ["age", "workclass", "fnlwgt", "education", "education_num",
                    "marital_status", "occupation", "relationship", "race",
                    "sex",
                    "capital_gain", "capital_loss", "hours_per_week",
                    "native_country", "income"]
    income_df = sqlContext.read.format("com.databricks.spark.csv").options(
        header="false", inferschema="true").load(csv_path)
    for old_name, new_name in zip(income_df.columns, column_names):
        income_df = income_df.withColumnRenamed(old_name, new_name)
        income_df = income_df.dropna()
    # print(income_df.dtypes)
    print("Load csv file correctly.")
    return income_df
```

2. As for the data preprocessing part, please see the python script in detail.

```python
def preprocess(data_frame):
    category_columns = ["workclass", "education", "marital_status",
                        "occupation", "relationship", "race", "sex",
                        "native_country", "income"]
    index_columns = [col + "_index" for col in category_columns]
    vec_columns = [col + "_vec" for col in category_columns]
    for col in category_columns:
        stringIndexer = StringIndexer(inputCol=col,
                                      outputCol=col + "_index",
                                      handleInvalid='error')
        model = stringIndexer.fit(data_frame)
        data_frame = model.transform(data_frame)
        data_frame = data_frame.drop(col)
    index_columns.pop(-1)
    vec_columns.pop(-1)

    ohe = OneHotEncoderEstimator(inputCols=index_columns,
                                 outputCols=vec_columns)
    ohe_model = ohe.fit(data_frame)
    ohe_df = ohe_model.transform(data_frame)
    ohe_df = ohe_df.drop(*index_columns)
    # ohe_df.show()
    cols = ohe_df.columns
    cols.remove("income_index")
    vector_assembler = VectorAssembler(inputCols=cols, outputCol="features")
    vdata_frame = vector_assembler.transform(ohe_df)
    vdata_frame = vdata_frame.drop(*cols)
    # vdata_frame.show()
    print("Preprocess input data correctly.")
    return vdata_frame
```

3. As for the data modelling part, please see the python script in detail.

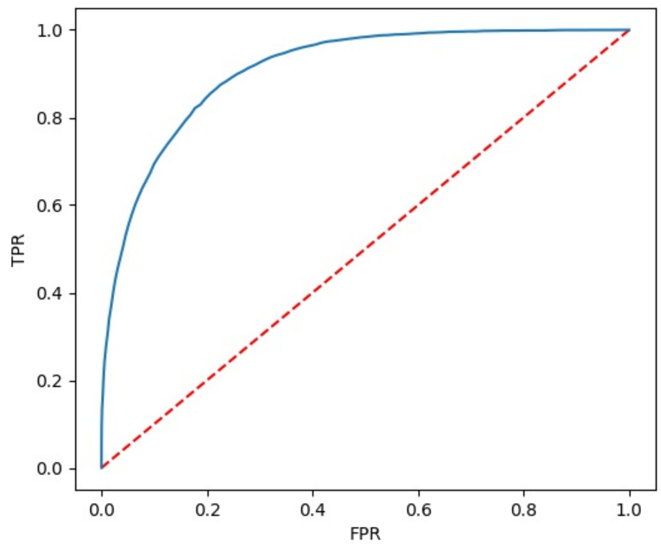


Figure 5: ROC curve of training process

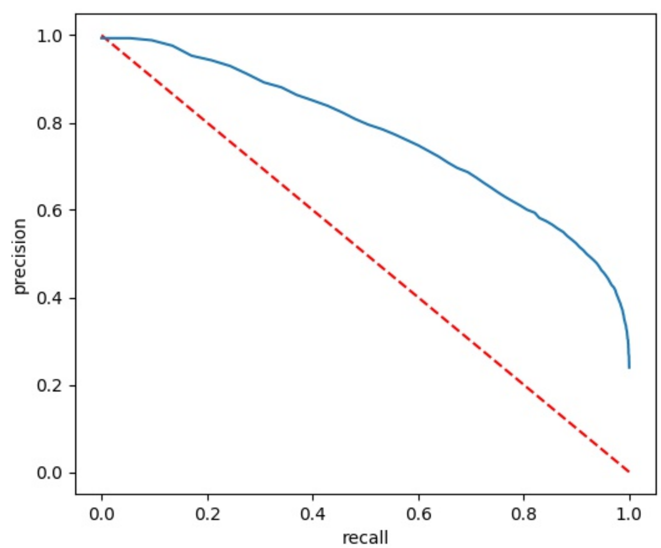


Figure 6: Precision-Recall curve of training process

4. Evaluation part:

```
Training:
Training Area under ROC = 0.9084271679599486
Training Accuracy = 0.8558104912864524
```

```

Evaluation:
Testing Area under ROC = 0.9076680514568874
Testing Accuracy = 0.8445952895196955
Testing Confusion Matrix:
[[6820.  528.]
 [ 983. 1392.]]
Total cost 29.132921s
Done!
```

Here we can see value of area under ROC, accuracy, and confusion matrix on test data set.

Here is whole code for part II.

```python
import numpy as np
import time
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext
from pyspark.ml.feature import OneHotEncoderEstimator, StringIndexer, \
    VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.mllib.classification import LogisticRegressionWithLBFGS
from pyspark.mllib.evaluation import BinaryClassificationMetrics, \
    MulticlassMetrics
from matplotlib import pyplot as plt

CSV_PATH = "data/adult.data.csv"


def load(sqlContext, csv_path):
    column_names = ["age", "workclass", "fnlwgt", "education", "education_num",
                    "marital_status", "occupation", "relationship", "race",
                    "sex",
                    "capital_gain", "capital_loss", "hours_per_week",
                    "native_country", "income"]
    income_df = sqlContext.read.format("com.databricks.spark.csv").options(
        header="false", inferschema="true").load(csv_path)
    for old_name, new_name in zip(income_df.columns, column_names):
        income_df = income_df.withColumnRenamed(old_name, new_name)
        income_df = income_df.dropna()
    # print(income_df.dtypes)
    print("Load csv file correctly.")
    return income_df

```

```python

def preprocess(data_frame):
    category_columns = ["workclass", "education", "marital_status",
                        "occupation", "relationship", "race", "sex",
                        "native_country", "income"]
    index_columns = [col + "_index" for col in category_columns]
    vec_columns = [col + "_vec" for col in category_columns]
    for col in category_columns:
        stringIndexer = StringIndexer(inputCol=col,
                                      outputCol=col + "_index",
                                      handleInvalid='error')
        model = stringIndexer.fit(data_frame)
        data_frame = model.transform(data_frame)
        data_frame = data_frame.drop(col)
    index_columns.pop(-1)
    vec_columns.pop(-1)

    ohe = OneHotEncoderEstimator(inputCols=index_columns,
                                 outputCols=vec_columns)
    ohe_model = ohe.fit(data_frame)
    ohe_df = ohe_model.transform(data_frame)
    ohe_df = ohe_df.drop(*index_columns)
    # ohe_df.show()
    cols = ohe_df.columns
    cols.remove("income_index")
    vector_assembler = VectorAssembler(inputCols=cols, outputCol="features")
    vdata_frame = vector_assembler.transform(ohe_df)
    vdata_frame = vdata_frame.drop(*cols)
    # vdata_frame.show()
    print("Preprocess input data correctly.")
    return vdata_frame


def plot_roc(FPR, TPR, img_path):
    fig = plt.figure(figsize=(6, 5))
    plt.plot([0, 1], [0, 1], 'r--')
    plt.plot(FPR, TPR)
    plt.xlabel('FPR')
    plt.ylabel('TPR')
    fig.savefig(img_path)


def plot_pr(recall, precision, img_path):
    fig = plt.figure(figsize=(6, 5))
    plt.plot([0, 1], [1, 0], 'r--')
```

```
    plt.plot(recall, precision)
    plt.xlabel('recall')
    plt.ylabel('precision')
    fig.savefig(img_path)


def main():
    start = time.time()
    conf = SparkConf().setMaster("local").setAppName("income")
    sc = SparkContext(conf=conf)
    sqlContext = SQLContext(sc)
    income_df = load(sqlContext, csv_path=CSV_PATH)
    # income_df.show()
    # print(income_df.dtypes)
    # print(income_df.count())

    features_df = preprocess(data_frame=income_df)

    # train, test split
    train_df, test_df = features_df.randomSplit([7.0, 3.0], 100)

    # logistic regression

    income_lr = LogisticRegression(featuresCol="features",
                                   labelCol="income_index",
                                   regParam=0.0, elasticNetParam=0.0,
                                   maxIter=200)
    income_model = income_lr.fit(train_df)

    # modeling
    print("Training:")
    training_summary = income_model.summary
    training_FPR = training_summary.roc.select('FPR').collect()
    training_TPR = training_summary.roc.select('TPR').collect()
    plot_roc(training_FPR, training_TPR, "pic/training_roc.jpg")

    training_recall = training_summary.pr.select('recall').collect()
    training_precision = training_summary.pr.select('precision').collect()
    # Area under ROC curve
    print("Training Area under ROC = %s" % training_summary.areaUnderROC)
    # accuracy
    print("Training Accuracy = %s" % training_summary.accuracy)
    plot_pr(training_recall, training_precision, "pic/training_pr.jpg")

    # evaluation
```

```
    print()
    print("Evaluation:")
    pred_df = income_model.transform(test_df).select("prediction",
                                                     "income_index")
    raw_pred_df = income_model.transform(test_df).select("probability",
                                                     "income_index"
                                                     ).rdd.map(
        lambda l: (float(l[0][1]), l[1]))
    metrics = BinaryClassificationMetrics(raw_pred_df)
    # Area under ROC curve
    print("Testing Area under ROC = %s" % metrics.areaUnderROC)
    # accuracy
    metrics = MulticlassMetrics(pred_df.rdd)
    print("Testing Accuracy = %s" % metrics.accuracy)

    # confusion matrix
    print("Testing Confusion Matrix:")
    print(metrics.confusionMatrix().toArray())
    print("Total cost %fs" % (time.time() - start))
    print("Done!")


if __name__ == '__main__':
    main()
```