HW1_JingQian_Q1

Jing Qian (jq2282@columbia.edu)

October 4, 2019

1 Install Spark

```
In [0]: # Install latest version of spark. If error, check the latest and replace "spark-2.4.4"
        !apt-get install openjdk-8-jdk-headless -qq > /dev/null
        !wget -q https://www-us.apache.org/dist/spark/spark-2.4.4/spark-2.4.4-bin-hadoop2.7.tgz
        !tar xf spark-2.4.4-bin-hadoop2.7.tgz
        !pip install -q findspark
        import os
        os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
        os.environ["SPARK_HOME"] = "/content/spark-2.4.4-bin-hadoop2.7"
        import findspark
        findspark.init()
```

```
In [115]: import numpy as np
          import pandas as pd

          from google.colab import drive
          drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

2 Q1. Implement iterative K-means in Spark

Based on the kmeans.py provided

```
In [0]: import operator
        import sys
        from pyspark import SparkConf, SparkContext
        import numpy as np
        import matplotlib.pyplot as plt
        from scipy import linalg
```

```
In [0]: # Macros.
        MAX_ITER = 20
        DATA_PATH = "/content/gdrive/My Drive/BigData/q1/data.txt"
        C1_PATH = "/content/gdrive/My Drive/BigData/q1/c1.txt"
        C2_PATH = "/content/gdrive/My Drive/BigData/q1/c2.txt"
        NORM = 2
```

```
In [0]: # Load data (corresponding to the 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()
```

```
In [0]: # 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
```

```
In [0]: def dist(centroid, p, norm):
            """
            Compute closest centroid for a given point.
            Args:
                centroid (numpy.ndarray): centroid of the cluster p belongs to
                p (numpy.ndarray): input point
                norm (int): 1 or 2
            Returns:
                float: the distance between centroid and p.
            """
            res = 0
            if norm == 1:
```

```
            res = linalg.norm(p - centroid, norm)
          elif norm == 2:
            res = linalg.norm(p - centroid, norm) ** 2
          return res
```

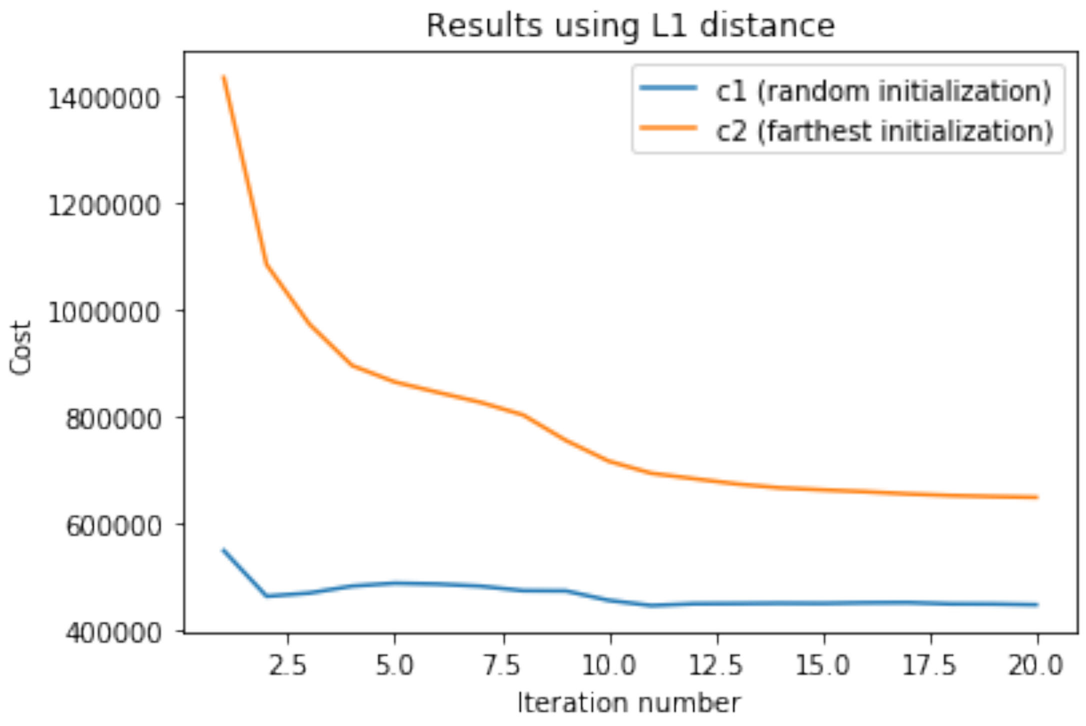
```
In [0]: # K-means clustering
        def kmeans(data, centroids, norm=2):
            """
            Conduct k-means clustering given data and centroid.
            This is the basic version of k-means, you might need more
            code to record cluster assignment to plot TSNE, and more
            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))
                list: a list of centroids
                loss: a list of within-cluster cost
            """
            # iterative k-means
            loss = []
            for _ in range(MAX_ITER):
                # Transform each point to a combo of point, closest centroid, count=1
                # point -> (closest_centroid, (point, 1))
                data_trans = data.map(lambda p:(closest(p, centroids,norm),(p,1)))

                # Compute the loss
                data_dist = data_trans.map(lambda p: dist(centroids[p[0]], p[1][0], norm))
                loss.append(sum(data_dist.collect()))

                # Re-compute cluster center
                # For each cluster center (key), aggregate its values
                # by summing up points and count
                clusters = data_trans.reduceByKey(lambda p1_c, p2_c: (p1_c[0]+p2_c[0], p1_c[1]+p

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

            return data_trans, centroids, loss
```

2.1 (1). Within-cluster cost using L1 distance.

Run clustering on data.txt with c1.txt and c2.txt as initial centroids and use L1 distance as similarity measurement. Compute and plot the within-cluster cost for each iteration.

```
In [0]: data11, centroids11, loss11 = kmeans(data, centroids1, norm=1)
        data21, centroids21, loss21 = kmeans(data, centroids2, norm=1)
```

```
In [14]: import matplotlib.pyplot as plt
         x = np.arange(1,21)
         plt.plot(x,loss11, label='c1 (random initialization)')
         plt.plot(x,loss21, label='c2 (farthest initialization)')
         plt.title('Results using L1 distance')
         plt.xlabel('Iteration number')
         plt.ylabel('Cost')
         plt.legend()
         plt.savefig('/content/gdrive/My Drive/BigData/q1/Q1_1.png')
```

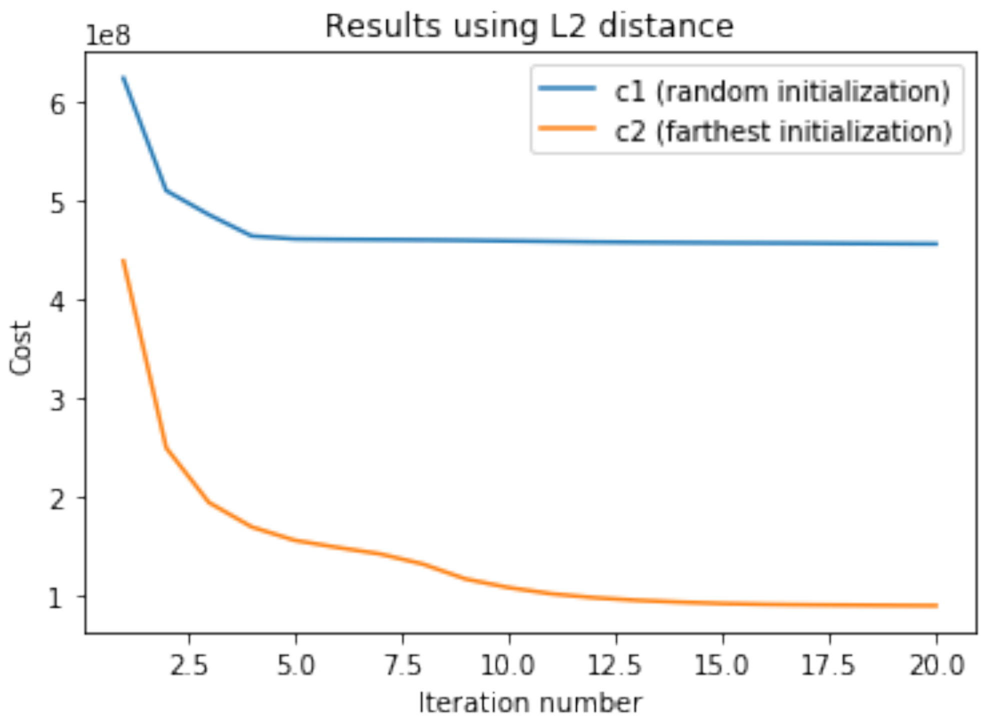


2.2 (2). Within-cluster cost using L2 distance.

Run clustering on data.txt with c1.txt and c2.txt as initial centroids and use L2 distance as similarity measurement. Compute and plot the within-cluster cost for each iteration.

```
In [0]: data12, centroids12, loss12 = kmeans(data, centroids1, norm=2)
        data22, centroids22, loss22 = kmeans(data, centroids2, norm=2)
```

```
In [13]: import matplotlib.pyplot as plt
         x = np.arange(1,21)
         plt.plot(x,loss12, label='c1 (random initialization)')
         plt.plot(x,loss22, label='c2 (farthest initialization)')
         plt.title('Results using L2 distance')
         plt.xlabel('Iteration number')
         plt.ylabel('Cost')
         plt.legend()
         plt.savefig('/content/gdrive/My Drive/BigData/q1/Q1_2.png')
```



2.3 (3) Visualize clustering result of (2) by T-SNE

```
In [0]: from sklearn.manifold import TSNE
```

```
In [0]: data12_np = np.array(data12.collect())
        keys12 = data12_np[:,0]
```

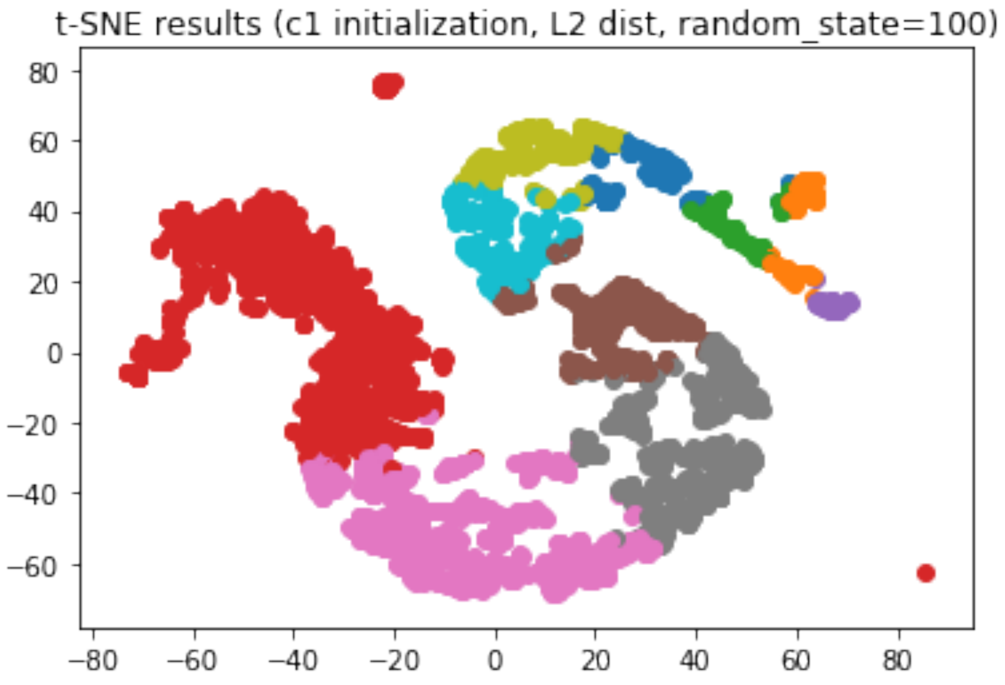
```
In [110]: values12 = []
          for i in data12_np:
            values12.append(i[1][0])
          values12_embedded = TSNE(n_components=2, random_state=100).fit_transform(values12)
          print(np.shape(values12), np.shape(values12_embedded))
```

```
(4601, 58) (4601, 2)
```

```
In [122]: colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd',
           '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf']
```

```
         for i in range(len(keys)):
           plt.scatter(values12_embedded[i][0], values12_embedded[i][1], color=colors[keys[i]])
         plt.title("t-SNE results (c1 initialization, L2 dist, random_state=100)")
```

```
Out[122]: Text(0.5, 1.0, 't-SNE results (c1 initialization, L2 dist, random_state=100)')
```

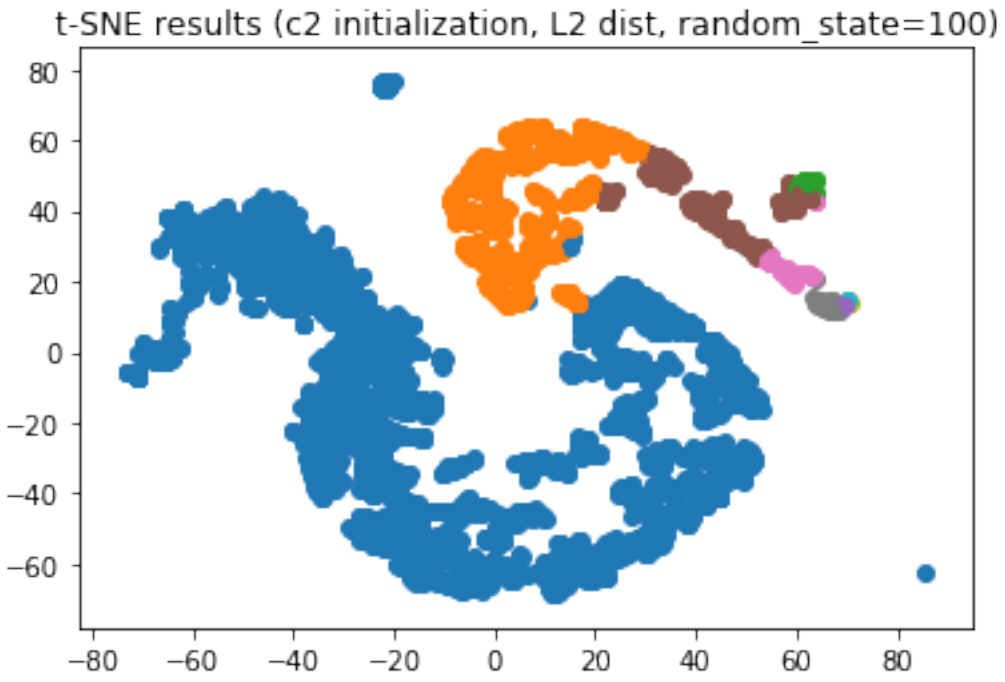


```
In [123]: data22_np = np.array(data22.collect())
          keys22 = data22_np[:,0]
          values22 = []
          for i in data22_np:
            values22.append(i[1][0])
          values22_embedded = TSNE(n_components=2, random_state=100).fit_transform(values22)
          print(np.shape(values22), np.shape(values22_embedded))
```

```
(4601, 58) (4601, 2)
```

```
In [124]: for i in range(len(keys22)):
            plt.scatter(values22_embedded[i][0], values22_embedded[i][1], color=colors[keys22[i]
          plt.title("t-SNE results (c2 initialization, L2 dist, random_state=100)")
```

```
Out[124]: Text(0.5, 1.0, 't-SNE results (c2 initialization, L2 dist, random_state=100)')
```



2.4 (4) For L2 and L1 distance, are random initialization of K-means using c1.txt better than initialization using c2.txt in terms of cost? Explain your reasoning.

For L1 distance, the random initialization of K-means is much better than that using c2.txt in terms of cost. Although the cost of farthest initialization decreases with the increasing of iterate times, it is far above that from random initialization. I suppose that the definition of "farthest" in farthest initialization refers to the L2 distance between points and hence it may not behave well on the L1 distance clustering. Also, farthest initialization is sensitive to outliers, which may contributes to the high cost of L1 distance clustering cost. On the contrary, random initialization lead to a lower cost preventing such problems.

For L2 distance, the random initialization of K-means is worse than using c2.txt in terms of cost. Although both costs decrease with the increasing of iterate times, the cost of c1 is all above c2 in the plot. As previous analysis, considering the farthest initialization defined as points with farthest L2 distance, the clusters in the first running already tended to spread apart. And so the farthest initialization using c2.txt has lower cost for L2 distance than random initialization.

2.5 (5) What is the time complexity of the iterative K-means?

The iterative K-means includes three layers of loops: the outer loop iterates MAX_ITER times, the intermediate loop iterates on all points in the dataset and the inner loop iterates over k clusters/centroids. So the time complexity of the iterative K-means is O(k*MAX_ITER*#points).

```
In [0]:
```

HW1_JingQian_Q2

October 3, 2019

1 Load Spark

```
In [0]: # Install latest version of spark. If error, check the latest and replace "spark-2.4.4"
        !apt-get install openjdk-8-jdk-headless -qq > /dev/null
        !wget -q https://www-us.apache.org/dist/spark/spark-2.4.4/spark-2.4.4-bin-hadoop2.7.tgz
        !tar xf spark-2.4.4-bin-hadoop2.7.tgz
        !pip install -q findspark
        import os
        os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
        os.environ["SPARK_HOME"] = "/content/spark-2.4.4-bin-hadoop2.7"
        import findspark
        findspark.init()
```

```
In [0]: #The entry point to using Spark SQL is an object called SparkSession.
        #It initiates a Spark Application which all the code for that Session will run on.
        from pyspark.sql import SparkSession
        spark = SparkSession.builder \
            .master("local[*]") \
            .appName("Learning_Spark") \
            .getOrCreate()
```

```
In [0]: import numpy as np
        import pandas as pd

        from google.colab import drive
        drive.mount('/content/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6b

Enter your authorization code:
ůůůůůůůůůů
Mounted at /content/gdrive
```

2 Q2. Binary classification with Spark MLlib

2.1 (1). Data loading

Read the csv file into a Dataframe. You could set "inferschema" to true and rename the columns with the following information: "age", "workclass", "fnlwgt","education", "education_num", "marital_status", "occupation", "relationship","race", "sex", "capital_gain", "capital_loss", "hours_per_week", "native_country","income".

```
In [0]: DATA_PATH = "/content/gdrive/My Drive/BigData/q1/adult_data.csv"
```

```
In [74]: data = spark.read.csv(DATA_PATH,inferSchema=True, header=False)
         print(data.count(),len(data.columns))
         data.show(5)
```

```
32561 15
+---+-----------------+--------+----------+----+-------------------+------------------+---------
|_c0|              _c1|     _c2|       _c3| _c4|                _c5|               _c6|
+---+-----------------+--------+----------+----+-------------------+------------------+---------
| 39|        State-gov| 77516.0| Bachelors|13.0|      Never-married|      Adm-clerical| Not-in-f
| 50| Self-emp-not-inc| 83311.0| Bachelors|13.0| Married-civ-spouse|   Exec-managerial|       Hu
| 38|          Private|215646.0|   HS-grad| 9.0|           Divorced| Handlers-cleaners| Not-in-f
| 53|          Private|234721.0|      11th| 7.0| Married-civ-spouse| Handlers-cleaners|       Hu
| 28|          Private|338409.0| Bachelors|13.0| Married-civ-spouse|    Prof-specialty|
+---+-----------------+--------+----------+----+-------------------+------------------+---------
only showing top 5 rows
```

```
In [75]: col_names = ["age", "workclass", "fnlwgt", "education", "education_num", "marital_statu
                      "occupation", "relationship", "race", "sex", "capital_gain", "capital_loss
                      "hours_per_week", "native_country", "income"]
         print(len(col_names))
         data = data.toDF(*col_names)
         data.show(5)
```

```
15
+---+-----------------+--------+----------+-------------+-------------------+------------------+
|age|        workclass|  fnlwgt| education|education_num|     marital_status|        occupation|
+---+-----------------+--------+----------+-------------+-------------------+------------------+
| 39|        State-gov| 77516.0| Bachelors|         13.0|      Never-married|      Adm-clerical|
| 50| Self-emp-not-inc| 83311.0| Bachelors|         13.0| Married-civ-spouse|   Exec-managerial|
| 38|          Private|215646.0|   HS-grad|          9.0|           Divorced| Handlers-cleaners|
| 53|          Private|234721.0|      11th|          7.0| Married-civ-spouse| Handlers-cleaners|
| 28|          Private|338409.0| Bachelors|         13.0| Married-civ-spouse|    Prof-specialty|
+---+-----------------+--------+----------+-------------+-------------------+------------------+
only showing top 5 rows
```

```
In [76]: data.printSchema()

root
 |-- age: integer (nullable = true)
 |-- workclass: string (nullable = true)
 |-- fnlwgt: double (nullable = true)
 |-- education: string (nullable = true)
 |-- education_num: double (nullable = true)
 |-- marital_status: string (nullable = true)
 |-- occupation: string (nullable = true)
 |-- relationship: string (nullable = true)
 |-- race: string (nullable = true)
 |-- sex: string (nullable = true)
 |-- capital_gain: double (nullable = true)
 |-- capital_loss: double (nullable = true)
 |-- hours_per_week: double (nullable = true)
 |-- native_country: string (nullable = true)
 |-- income: string (nullable = true)
```

2.2 (2). Data preprocessing

Convert the categorical variables into numeric variables with ML Pipelines and Feature Transformers . You will probably need OneHotEncoderEstimator, StringIndexer, and VectorAssembler. Split your data into training set and test set with ratio of 70% and 30% and set the seed to 100.

Reference: https://towardsdatascience.com/machine-learning-with-pyspark-and-mllib-solving-a-binary-classification-problem-96396065d2aa

```
In [77]: train, test = data.randomSplit([0.7, 0.3], seed = 100)
         print("Training Dataset Count: " + str(train.count()))
         print("Test Dataset Count: " + str(test.count()))

Training Dataset Count: 22838
Test Dataset Count: 9723
```

```
In [0]: from pyspark.ml.feature import OneHotEncoderEstimator, StringIndexer, VectorAssembler
        categoricalColumns = ['workclass','education','marital_status','occupation','relationshi
                              'race','sex','native_country']
        stages = []
        for categoricalCol in categoricalColumns:
            stringIndexer = StringIndexer(inputCol = categoricalCol, outputCol = categoricalCol
            encoder = OneHotEncoderEstimator(inputCols=[stringIndexer.getOutputCol()], outputCol
            stages += [stringIndexer, encoder]
        label_stringIdx = StringIndexer(inputCol = 'income', outputCol = 'label')
        stages += [label_stringIdx]
        numericCols = ['age','fnlwgt','education_num','capital_gain','capital_loss','hours_per_w
        assemblerInputs = [c + "classVec" for c in categoricalColumns] + numericCols
        assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")
        stages += [assembler]
```

```
In [79]: from pyspark.ml import Pipeline
         pipeline = Pipeline(stages = stages)
         pipelineModel = pipeline.fit(train)
         train = pipelineModel.transform(train)

         selectedCols = ['label', 'features'] + col_names
         train = train.select(selectedCols)
         train.printSchema()
```

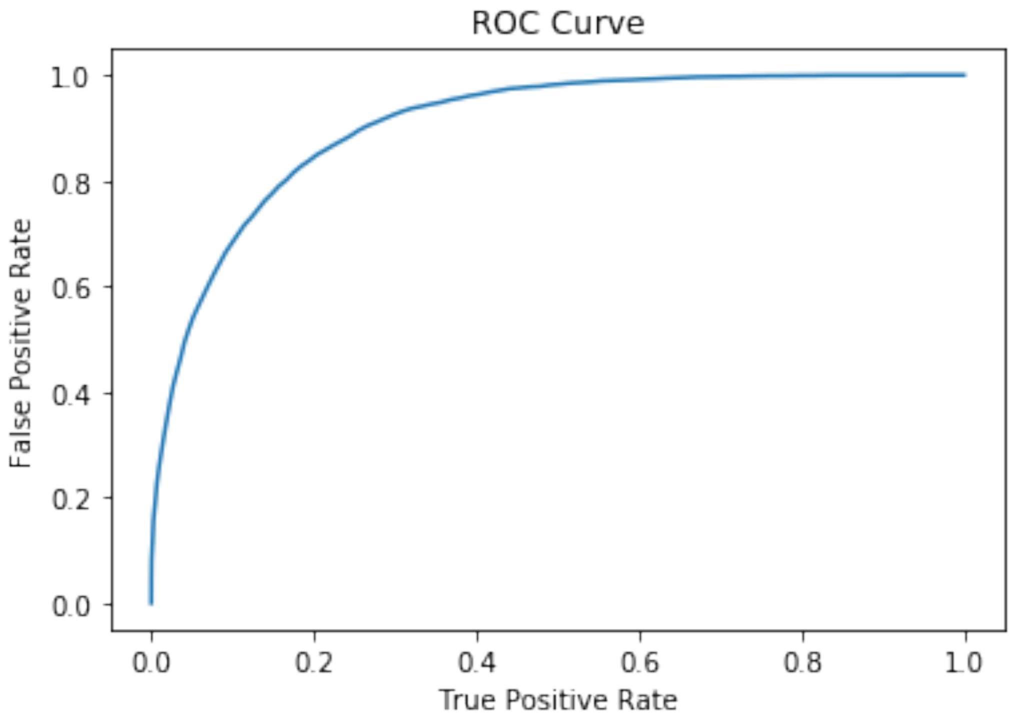
```
root
 |-- label: double (nullable = false)
 |-- features: vector (nullable = true)
 |-- age: integer (nullable = true)
 |-- workclass: string (nullable = true)
 |-- fnlwgt: double (nullable = true)
 |-- education: string (nullable = true)
 |-- education_num: double (nullable = true)
 |-- marital_status: string (nullable = true)
 |-- occupation: string (nullable = true)
 |-- relationship: string (nullable = true)
 |-- race: string (nullable = true)
 |-- sex: string (nullable = true)
 |-- capital_gain: double (nullable = true)
 |-- capital_loss: double (nullable = true)
 |-- hours_per_week: double (nullable = true)
 |-- native_country: string (nullable = true)
 |-- income: string (nullable = true)
```

2.3 (3). Modelling

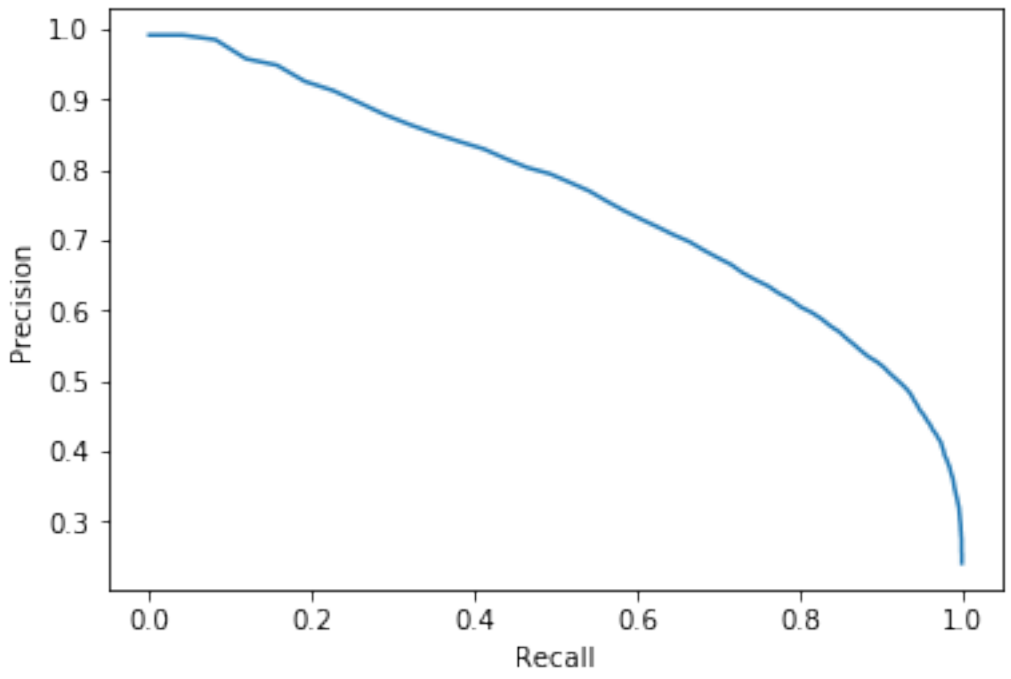
Train a logistic regression model with train set. Learn more about models provide in Spark MLlib here . After training, plot ROC curve and Precision-Recall curve of your training process.

```
In [0]: from pyspark.ml.classification import LogisticRegression
        lr = LogisticRegression(featuresCol = 'features', labelCol = 'label', maxIter=10)
        lrModel = lr.fit(train)
```

```
In [81]: import matplotlib.pyplot as plt
         trainingSummary = lrModel.summary
         roc = trainingSummary.roc.toPandas()
         plt.plot(roc['FPR'],roc['TPR'])
         plt.ylabel('False Positive Rate')
         plt.xlabel('True Positive Rate')
         plt.title('ROC Curve')
         plt.show()
         print('Training set areaUnderROC: ' + str(trainingSummary.areaUnderROC))

         pr = trainingSummary.pr.toPandas()
         plt.plot(pr['recall'],pr['precision'])
         plt.ylabel('Precision')
         plt.xlabel('Recall')
         plt.show()
```



```
Training set areaUnderROC: 0.9056654937412549
```



2.4 (4). Evaluation

Apply your trained model on the test set. Provide the value of area under ROC, accuracy, and confusion matrix. You should expect the accuracy to be around 85%.

```
In [82]: test = pipelineModel.transform(test)
         test = test.select(selectedCols)

         predictions = lrModel.transform(test)

         from pyspark.ml.evaluation import BinaryClassificationEvaluator
         evaluator = BinaryClassificationEvaluator()
         print('Test Area Under ROC', evaluator.evaluate(predictions))
```

```
Test Area Under ROC 0.9027382028865563
```

```
In [84]: from pyspark.ml.evaluation import MulticlassClassificationEvaluator

         # Select (prediction, true label) and compute test error
         evaluator = MulticlassClassificationEvaluator(
             labelCol="label", predictionCol="prediction", metricName="accuracy")
         accuracy = evaluator.evaluate(predictions)
         print("Test accuracy: ", accuracy)
```

```
Test accuracy:  0.8484006993726216
```

```
In [87]: from sklearn.metrics import confusion_matrix
         y_true = test.select('label').collect()
         y_pred = predictions.select('prediction').collect()
         cnf_matrix = confusion_matrix(y_true, y_pred)
         cnf_matrix
```

```
Out[87]: array([[6860,  530],
                [ 944, 1389]])
```