HW1_JingQian_Q1

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
```

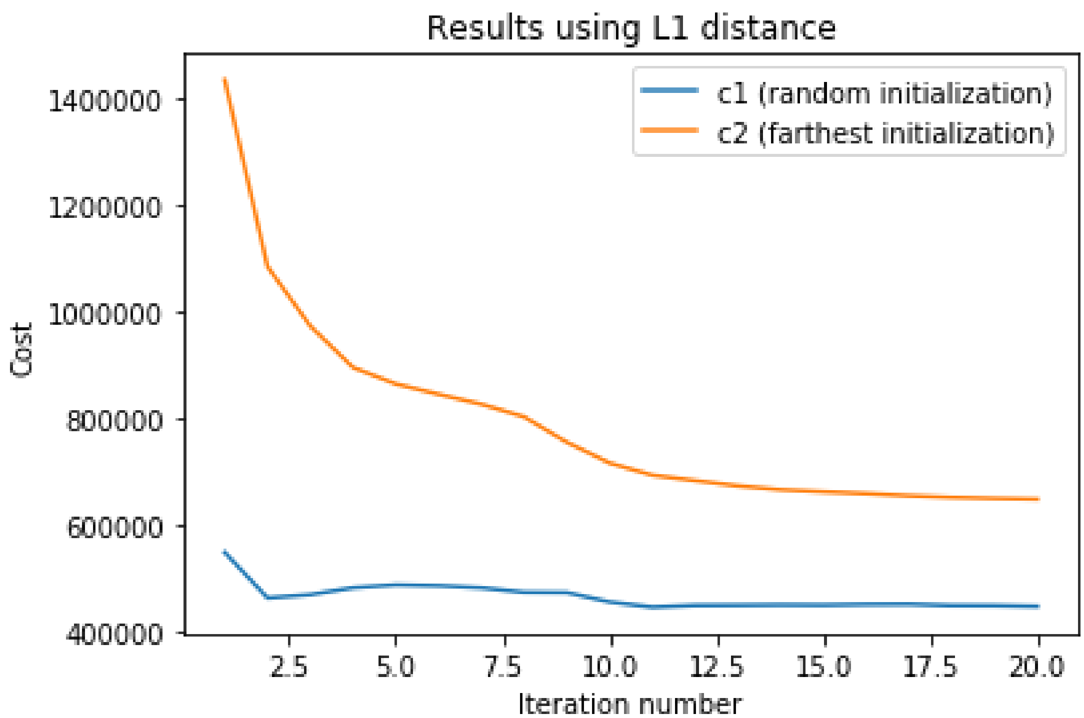
In [0]:
```
# K-means clustering
def kmeans(data, centroids, norm=2):
    """
    Conduct k-means clustering given data and centroid.
    This is the basic version of k-means, you might need more
    code to record cluster assignment to plot TSNE, and more
    data structure to record cost.
    Args:
        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')
```



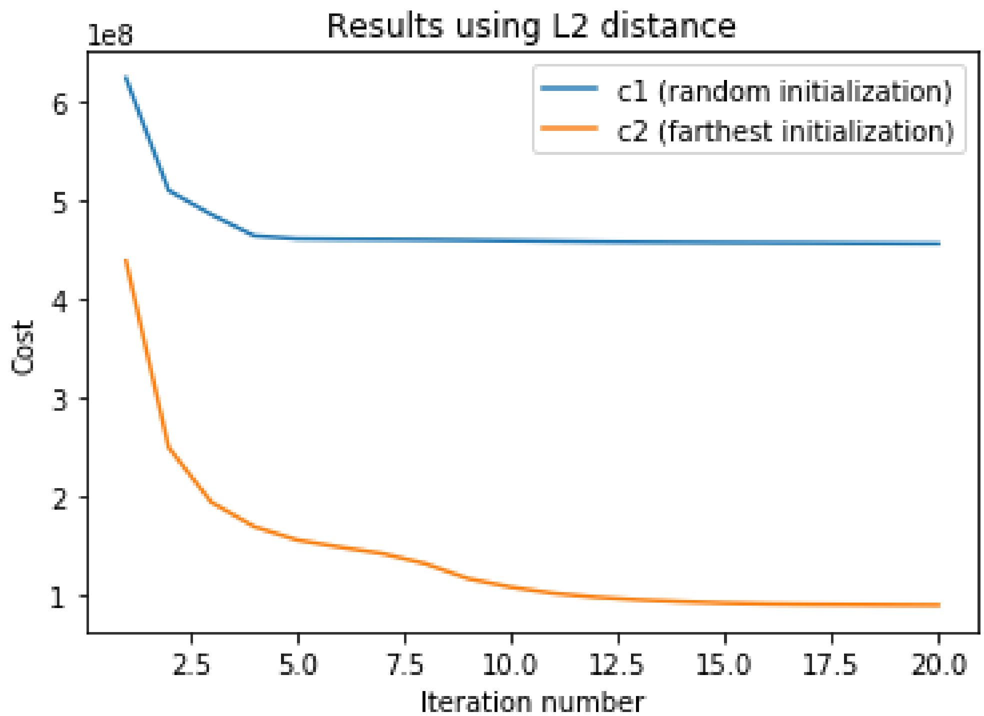
2.2 (2). Within-cluster cost using L2 distance.

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]
```

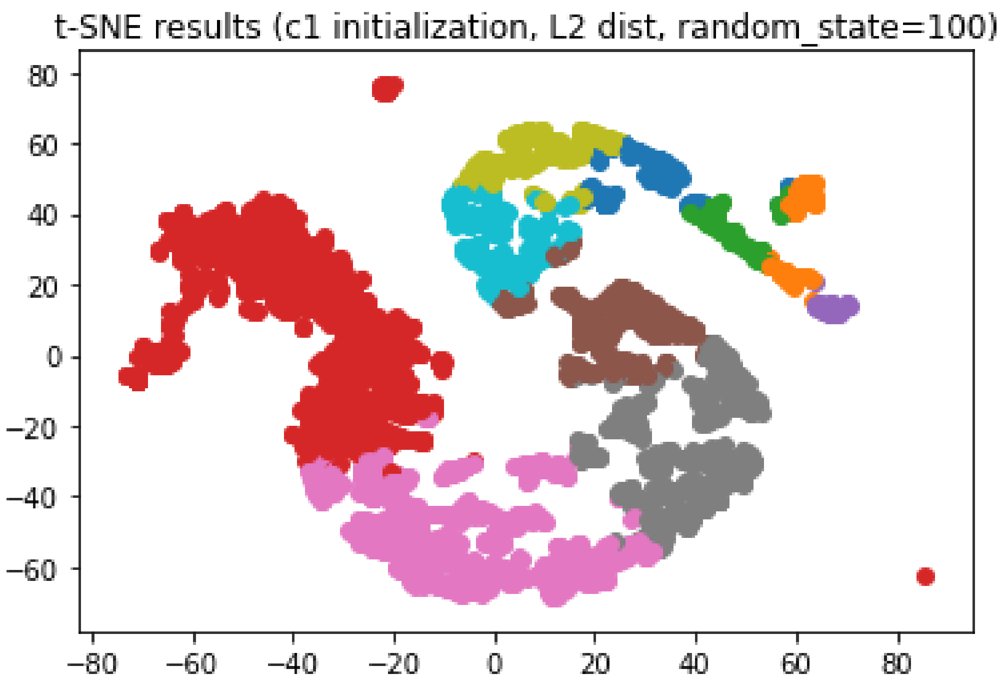
```
In [110]: values12 = []
          for i in data12_np:
            values12.append(i[1][0])
          values12_embedded = TSNE(n_components=2, random_state=100).fit_transform(values12)
          print(np.shape(values12), np.shape(values12_embedded))
```

```
(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)')

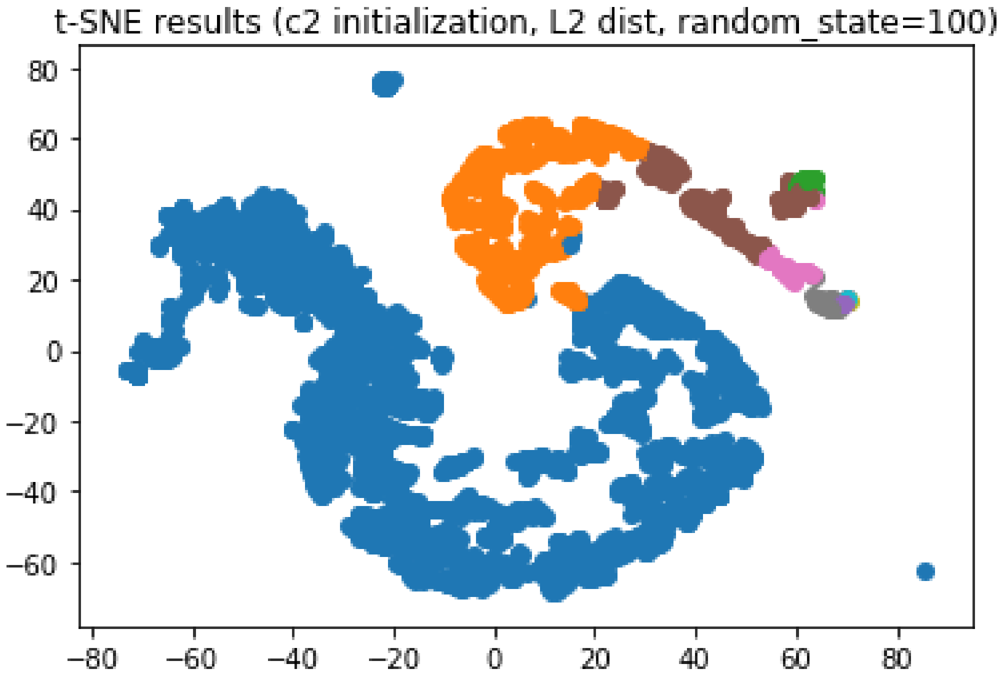


In [123]:
```
data22_np = np.array(data22.collect())
keys22 = data22_np[:,0]
values22 = []
for i in data22_np:
  values22.append(i[1][0])
values22_embedded = TSNE(n_components=2, random_state=100).fit_transform(values22)
print(np.shape(values22), np.shape(values22_embedded))
```

(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]:
```