Data Intensive Computing – Handout 11

Spark: Transitive Closure

spark/transitive_closure.py

```
import sys
from random import Random

from pyspark import SparkContext

num_edges = 3000
num_vertices = 500
rand = Random(42)

def generate_graph():
    edges = set()
    while len(edges) < num_edges:
        src = rand.randrange(0, num_vertices)
        dst = rand.randrange(0, num_vertices)
        if src != dst:
            edges.add((src, dst))
    return edges

if __name__ == "__main__":
    if len(sys.argv) == 1:
        print >> sys.stderr, "Usage: transitive_closure <master> [<slices>]"
        exit(1)
    sc = SparkContext(sys.argv[1], "TransitiveClosure")
    slices = int(sys.argv[2]) if len(sys.argv) > 2 else 1
    closure = sc.parallelize(generate_graph(), slices).cache()

    # Linear transitive closure: each round grows paths by one edge,
    # by joining the graph's edges with the already-discovered paths.
    # e.g. join the path (y, z) from the TC with the edge (x, y) from
    # the graph to obtain the path (x, z).

    # Because join() joins on keys, the edges are stored in reversed order.
    edges = closure.map(lambda (x, y): (y, x)).cache()

    old_count = 0L
    new_count = closure.count()
    while new_count != old_count:
        # Perform the join, obtaining an RDD of (y, (z, x)) pairs,
        # then project the result to obtain the new (x, z) paths.
        new_edges = closure.join(edges).map(lambda (_, (a, b)): (b, a))
        new_closure = closure.union(new_edges).distinct().cache()

        old_count, new_count = new_count, new_closure.count()

        closure.unpersist()
        closure = new_closure
    closure.unpersist()
    edges.unpersist()

    print "TC has %i edges" % closure.count()
```

Spark: MLLib

Machine Learning library.

Binary Classification

mllib/classification.py

```
import numpy as np
import sys

from pyspark import SparkContext
from pyspark.mllib.classification import LogisticRegressionWithSGD

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print >> sys.stderr, "Usage: %s <master> <file>" % sys.argv[0]
        exit(1)
    sc = SparkContext(sys.argv[1], appName="Classification")

    # Load and parse the data
    data = sc.textFile(sys.argv[2])
    parsedData = data.map(lambda line:
        np.array([float(x) for x in line.split(' ')]))
    model = LogisticRegressionWithSGD.train(parsedData)

    # Build the model
    labelsAndPreds = parsedData.map(lambda point: (int(point.item(0)),
            model.predict(point.take(range(1, point.size)))))

    # Evaluating the model on training data
    trainErr = labelsAndPreds.filter(lambda (v, p): v != p).count() /
        float(parsedData.count())
    print("Training Error = " + str(trainErr))
```

- `NaiveBayes`
  - `train(cls, data, lambda=1.0)`
  - The resulting model has method `predict(self, x)`.
- `SVMWithSGD`
  - `train(cls, data, iterations=100, step=1.0, regParam=1.0, miniBatchFraction=1.0, initialWeights=None)`
  - The resulting model has method `predict(self, x)`.
- `LogisticRegressionWithSGD`
  - `train(cls, data, iterations=100, step=1.0, regParam=1.0, miniBatchFraction=1.0, initialWeights=None)`
  - The resulting model has method `predict(self, x)`.

Linear Regression

mllib/regression.py

```
import numpy as np
import sys

from pyspark import SparkContext
from pyspark.mllib.regression import LinearRegressionWithSGD

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print >> sys.stderr, "Usage: %s <master> <file>" % sys.argv[0]
        exit(1)
    sc = SparkContext(sys.argv[1], appName="Regression")

    # Load and parse the data
    data = sc.textFile(sys.argv[2])
    parsedData = data.map(lambda line:
        np.array([float(x) for x in line.replace(',', ' ').split(' ')]))

    # Build the model
    model = LinearRegressionWithSGD.train(parsedData)

    # Evaluate the model on training data
    valuesAndPreds = parsedData.map(lambda point: (point.item(0),
            model.predict(point.take(range(1, point.size)))))
    MSE = valuesAndPreds.map(lambda (v, p): (v - p)**2).reduce(
        lambda x, y: x + y)/valuesAndPreds.count()
    print("Mean Squared Error = " + str(MSE))
```

- `LinearRegressionWithSGD`
    - `train(cls, data, iterations=100, step=1.0, miniBatchFraction=1.0, initialWeights=None)`
    - The resulting model has method `predict(self, x)`.
- `LassoRegressionWithSGD`
    - `train(cls, data, iterations=100, step=1.0, regParam=1.0, miniBatchFraction=1.0, initialWeights=None)`
    - The resulting model has method `predict(self, x)`.
- `RidgeRegressionWithSGD`
    - `train(cls, data, iterations=100, step=1.0, regParam=1.0, miniBatchFraction=1.0, initialWeights=None)`
    - The resulting model has method `predict(self, x)`.

Clustering

mllib/clustering.py

```
from math import sqrt
import numpy as np
import sys

from pyspark import SparkContext
from pyspark.mllib.clustering import KMeans

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print >> sys.stderr, "Usage: %s <master> <file>" % sys.argv[0]
        exit(1)
    sc = SparkContext(sys.argv[1], appName="Clustering")

    # Load and parse the data
    data = sc.textFile(sys.argv[2])
    parsedData = data.map(lambda line:
        np.array([float(x) for x in line.split(' ')]))

    # Build the model (cluster the data)
    clusters = KMeans.train(parsedData, 2, maxIterations=10,
            runs=30, initializationMode="random")

    # Evaluate clustering by computing Within Set Sum of Squared Errors
    def error(point):
        center = clusters.centers[clusters.predict(point)]
        return sqrt(sum([x**2 for x in (point - center)]))

    WSSSE = parsedData.map(error).reduce(lambda x, y: x + y)
    print("Within Set Sum of Squared Error = " + str(WSSSE))
```

- `KMeans`
  - `train(cls, data, k, maxIterations=100, runs=1, initializationMode="k-means||")`
    - `initializatioinMode` can be either `random` or `k-means||`
  - The resulting model has method `predict(self, x)`.

Collaborative Filtering

mllib/collaborative_filtering.py

```
from math import sqrt
import numpy as np
import sys

from pyspark import SparkContext
from pyspark.mllib.recommendation import ALS

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print >> sys.stderr, "Usage: %s <master> <file>" % sys.argv[0]
        exit(1)
    sc = SparkContext(sys.argv[1], appName="CollaborativeFiltering")
```

```python
# Load and parse the data
data = sc.textFile(sys.argv[2])
ratings = data.map(lambda line: array([float(x) for x in line.split(',')]))

# Build the recommendation model using Alternating Least Squares
model = ALS.train(ratings, 1, 20)

# Evaluate the model on training data
testdata = ratings.map(lambda p: (int(p[0]), int(p[1])))
predictions = model.predictAll(testdata).map(lambda r: ((r[0], r[1]), r[2]))
ratesAndPreds = ratings.map(lambda r: ((r[0], r[1]), r[2])).join(predictions)
MSE = ratesAndPreds.map(lambda r: (r[1][0] - r[1][1])**2).reduce(
    lambda x, y: x + y)/ratesAndPreds.count()
print("Mean Squared Error = " + str(MSE))
```

- `ALS`
  - `train(cls, ratings, rank, iterations=5, lambda=0.01, blocks=-1)`
  - `trainImplicit(cls, ratings, rank, iterations=5, lambda=0.01, blocks=-1, alpha=0.01)`
  - The resulting model has methods
    - `predict(self, user, product)`
    - `predictAll(self, usersProducts)`

Tasks

Solve the following tasks. Solution for each task is a Spark source processing the Wikipedia source data and producing required results.

Simple Tokenizer

We will commonly need to split given text into words (called *tokens*). You can do so easily by using function `wordpunct_tokenize` from `nltk.tokenize` package, i.e. using the following import line at the beginning of you program:

```
from nltk.tokenize import wordpunct_tokenize
```

Wikipedia Data

The textual Wikipedia Data are avilable in HDFS:

- `/data/wiki-txt/cs`: Czech Wikipedia data (Sep 2009), 195MB, 124k articles
- `/data/wiki-txt/en`: English Wikipedia data (Sep 2009), 4.9GB, 2.9M articles

All data are encoded in UTF-8 and contain one particle per line. Article name is separated by `\t` character from the article content.

Task	*Points*	*Description*
`spark_unique_words`	2	Create a list of unique words used in the articles using Spark. Convert them to lowercase to ignore case.
`spark_anagrams`	2	Two words are anagrams if one is a letter permutation of the other (ignoring case). For a given input, find all anagram classes that contain at least A words. Output each anagram class on a separate line.
`spark_sort`	3	You are given data consisting of (31-bit integer, string data) pairs. These are available in plain text format: • `/data/numbers-txt/numbers-small`: 3MB • `/data/numbers-txt/numbers-medium`: 184MB • `/data/numbers-txt/numbers-large`: 916MB You can assume that the integers are uniformly distributed. Your task is to sort these data, comparing the key numerically and not lexicographically. The lines in the output must be the same as in the input, only in different order. Your solution should work for TBs of data. For that reason, you must use multiple machines. If your job is executed using m machines, the output consists of m files, which when concatenated would produce sorted (key, value) pairs. In other words, each of the output files contains sorted (integer, data) pairs and all keys in one file are either smaller or larger than in other file. Your solution should work for any number of machines specified.

Task	*Points*	*Description*
`spark_nonuniform_sort`	4	Improve the `spark_sort` to handle nonuniform data. You can use the following exponentially distributed data: • `/data/numbers-txt/nonuniform-small`: 3MB • `/data/numbers-txt/nonuniform-medium`: 160MB • `/data/numbers-txt/nonuniform-large`: 797MB Assume we want to produce m output files. One of the solutions is the following: • Go through the data and sample only a small fraction of the keys. • Find best $m-1$ separators using the sampled data. • Run the second pass using the computed separators.
`spark_inverted_index`	2	Compute inverted index in Spark – for every lowercased word from the articles, compute *(article name, ascending positions of occurrences as word indices)* pairs. The output should be a file with one word on a line in the following format: `word \t articleName \t spaceSeparatedOccurrences...` You will get 2 additional points if the articles will be numbered using consecutive integers. In that case, the output is ascending *(article id, ascending positions of occurrences as word indices)* pairs, together with a file containing list of articles representing this mapping (the article on line i is the article with id i).
`spark_no_references`	3	An article A is said to reference article B, if it contains B as a token (ignoring case). Run a Spark job which for each article B counts how many references for article B there exist in the whole wiki (summing references in a single article). You will get one extra point if the result is sorted by the number of references.
`spark_wordsim_index`	4	In order to implement word similarity search, compute for each form with at least three occurrences all *contexts* in which it occurs, including their number of occurrences. List the contexts in ascending order. Given N (either 1, 2, 3 or 4), the *context* of a form occurrence is N forms preceding this occurrence and N forms following this occurrence (ignore sentence boundaries, use empty words when article boundaries are reached). The output should be a file with one form on a line in the following format: `form \t context \t counts...`

<table>
<tr><th>Task</th><th>Points</th><th>Description</th></tr>
<tr><td><code>spark_wordsim_find</code></td><td>4</td><td>Let S be given natural number. Using the index created in <code>spark_wordsim_index</code>, find for each form S most similar forms. The similarity of two forms is computed using cosine similarity as $\frac{C_A \cdot C_B}{|C_A| \cdot |C_B|}$, where C_F is a vector of occurrences of form F contexts.
The output should be a file with one form on a line in the following format:
<code>form \t most similar form \t cosine similarity...</code></td></tr>
<tr><td><code>spark_kmeans</code></td><td>6</td><td>Implement K-means clustering algorithm as described on <code>http://en.wikipedia.org/wiki/K-means_clustering#Standard_algorithm</code>.
The user specifies number of iterations and the program run specified number of K-means clustering algorithm iterations.
You can use the following training data. Each line contains space separated coordinates of one points. The coordinates in one input naturally have the same dimension.
<table>
<tr><th>HDFS path</th><th>Points</th><th>Dimension</th><th>Clusters</th></tr>
<tr><td><code>/data/points-txt/small</code></td><td>10000</td><td>50</td><td>50</td></tr>
<tr><td><code>/data/points-txt/medium</code></td><td>100000</td><td>100</td><td>100</td></tr>
<tr><td><code>/data/points-txt/large</code></td><td>500000</td><td>200</td><td>200</td></tr>
</table>
You will get 2 additional points if the algorithm stops when none of the centroid positions change more than given ε.</td></tr>
</table>