Text Analysis and Entity Resolution

This assignment is the continuation of the previous assignment.

Part 4: Scalable ER

In the previous parts, we built a text similarity function and used it for small scale entity resolution. Our implementation is limited by its quadratic run time complexity, and is not practical for even modestly sized datasets. In this part, we will implement a more scalable algorithm and use it to do entity resolution on the full dataset.

Inverted Indices
To improve our ER algorithm from the earlier parts, we should begin by analyzing its running time. In particular, the algorithm above is quadratic in two ways. First, we did a lot of redundant computation of tokens and weights, since each record was reprocessed every time it was compared. Second, we made quadratically many token comparisons between records.

The first source of quadratic overhead can be eliminated with precomputation and look-up tables, but the second source is a little more tricky. In the worst case, every token in every record in one dataset exists in every record in the other dataset, and therefore every token makes a non-zero contribution to the cosine similarity. In this case, token comparison is unavoidably quadratic.

But in reality most records have nothing (or very little) in common. Moreover, it is typical for a record in one dataset to have at most one duplicate record in the other dataset (this is the case assuming each dataset has been de-duplicated against itself). In this case, the output is linear in the size of the input and we can hope to achieve linear running time.

An **inverted index** is a data structure that will allow us to avoid making quadratically many token comparisons. It maps each token in the dataset to the list of documents that contain the token. So, instead of comparing, record by record, each token to every other token to see if they match, we will use inverted indices to *look up* records that match on a particular token. In text search, a *forward* index maps documents in a dataset to the tokens they contain. An *inverted* index supports the inverse mapping. For this section, use the complete Google and Amazon datasets, not the samples.

(4a) Tokenize the full dataset
Tokenize each of the two full datasets for Google and Amazon.

>
```
# TODO: Replace <FILL IN> with appropriate code
amazonFullRecToToken = amazon.<FILL IN>
googleFullRecToToken = google.<FILL IN>
print('Amazon full dataset is %s products, Google full dataset is %s products' %
(amazonFullRecToToken.count(), googleFullRecToToken.count()))
Amazon full dataset is 1363 products, Google full dataset is 3226 products
```

(4b) Compute IDFs and TF-IDFs for the full datasets
We will reuse your previous code to compute IDF weights for the complete combined datasets. The steps you should perform are:

- Create a new `fullCorpusRDD` that contains the tokens from the full Amazon and Google datasets.
- Apply your `idfs` function to the `fullCorpusRDD`.

• Create a broadcast variable containing a dictionary of the IDF weights for the full dataset.
• For each of the Amazon and Google full datasets, create weight RDDs that map IDs/URLs to TF-IDF weighted token vectors.

>
```
# TODO: Replace <FILL IN> with appropriate code
fullCorpusRDD = <FILL IN>
idfsFull = idfs(fullCorpusRDD)
idfsFullCount = idfsFull.count()
print('There are %s unique tokens in the full datasets.' % idfsFullCount)
```
There are 17078 unique tokens in the full datasets.

```
# Recompute IDFs for full dataset
idfsFullWeights = <FILL IN>
idfsFullBroadcast = <FILL IN>

# Pre-compute TF-IDF weights.  Build mappings from record ID weight vector.
amazonWeightsRDD = <FILL IN>
googleWeightsRDD = <FILL IN>
print('There are %s Amazon weights and %s Google weights.' % (amazonWeightsRDD.count(),
                                                              googleWeightsRDD.count()))
```
There are 1363 Amazon weights and 3226 Google weights.

(4c) Compute Norms for the weights from the full datasets
We will reuse your previous code to compute norms of the IDF weights for the complete combined dataset. The steps you should perform are:
• Create two collections, one for each of the full Amazon and Google datasets, where IDs/URLs map to the norm of the associated TF-IDF weighted token vectors.
• Convert each collection into a broadcast variable, containing a dictionary of the norm of IDF weights for the full dataset.

>
```
# TODO: Replace <FILL IN> with appropriate code
amazonNorms = amazonWeightsRDD.<FILL IN>
amazonNormsBroadcast = <FILL IN>
googleNorms = googleWeightsRDD.<FILL IN>
googleNormsBroadcast = <FILL IN>

print(len(amazonNormsBroadcast.value))
```
1363
```
print(len(googleNormsBroadcast.value))
```
3226

(4d) Create inverted indicies from the full datasets
Build inverted indices of both data sources. The steps you should perform are:
• Create an invert function that given a pair of (ID/URL, TF-IDF weighted token vector), returns a list of pairs of (token, ID/URL). Recall that the TF-IDF weighted token vector is a Python dictionary with keys that are tokens and values that are weights.
• Use your invert function to convert the full Amazon and Google TF-IDF weighted token vector datasets into two RDDs where each element is a pair of a token and an ID/URL that contain that token. These are inverted indices.

>
```
# TODO: Replace <FILL IN> with appropriate code
```

```
def invert(record):
    """ Invert (ID, tokens) to a list of (token, ID)
    Args:
        record: a pair, (ID, token vector)
    Returns:
        pairs: a list of pairs of token to ID
    """
    <FILL IN>
    return (pairs)

print(invert((1, {'foo': 2})))
[('foo', 1)]

amazonInvPairsRDD = (amazonWeightsRDD
                     .<FILL IN>
                     .cache())

googleInvPairsRDD = (googleWeightsRDD
                     .<FILL IN>
                     .cache())

print('There are %s Amazon inverted pairs and %s Google inverted pairs.' % (amazonInvPairsRDD.count(),
                                                                            googleInvPairsRDD.count()))
There are 111387 Amazon inverted pairs and 77678 Google inverted pairs.
```

(4e) Identify common tokens from the full dataset

We are now in position to efficiently perform ER on the full datasets. Implement the following algorithm to build an RDD that maps a pair of (ID, URL) to a list of tokens they share in common:

- Using the two inverted indices (RDDs where each element is a pair of a token and an ID or URL that contains that token), create a new RDD that contains only tokens that appear in both datasets. This will yield an RDD of pairs of (token, iterable(ID, URL)).
- We need a mapping from (ID, URL) to token, so create a function that will swap the elements of the RDD you just created to create this new RDD consisting of ((ID, URL), token) pairs.
- Finally, create an RDD consisting of pairs mapping (ID, URL) to all the tokens the pair shares in common.

```
>
# TODO: Replace <FILL IN> with appropriate code
def swap(record):
    """ Swap (token, (ID, URL)) to ((ID, URL), token)
    Args:
        record: a pair, (token, (ID, URL))
    Returns:
        pair: ((ID, URL), token)
    """
    token = <FILL IN>
    keys = <FILL IN>
    return (keys, token)

commonTokens = (amazonInvPairsRDD
                .<FILL IN>
                .cache())

print('Found %d common tokens' % commonTokens.count())
Found 2441100 common tokens
```

(4f) Identify common tokens from the full dataset (cont.)
Use the data structures from parts **(4b)** and **(4e)** to build a dictionary to map record pairs to cosine similarity scores. The steps you should perform are:

- Create two broadcast dictionaries from the `amazonWeights` and `googleWeights` RDDs.
- Create a `fastCosinesSimilarity` function that takes in a record consisting of the pair ((Amazon ID, Google URL), tokens list) and computes the sum for each of the tokens in the token list of the products of the Amazon weight for the token times the Google weight for the token. The sum should then be divided by the norm for the Google URL and then divided by the norm for the Amazon ID. The function should return this value in a pair with the key being the (Amazon ID, Google URL). *Make sure you use broadcast variables you created for both the weights and norms*.
- Apply your `fastCosinesSimilarity` function to the common tokens from the full dataset.

>
```
# TODO: Replace <FILL IN> with appropriate code
amazonWeightsBroadcast = <FILL IN>
googleWeightsBroadcast = <FILL IN>

def fastCosineSimilarity(record):
    """ Compute Cosine Similarity using Broadcast variables
    Args:
        record: ((ID, URL), token)
    Returns:
        pair: ((ID, URL), cosine similarity value)
    """
    amazonRec = <FILL IN>
    googleRec = <FILL IN>
    tokens = <FILL IN>
    s = <FILL IN>
    value = <FILL IN>
    key = (amazonRec, googleRec)
    return (key, value)

similaritiesFullRDD = (commonTokens
                       .<FILL IN>
                       .cache())

print(similaritiesFullRDD.count())
2441100

similarityTest = similaritiesFullRDD.filter(lambda x: x[0][0] == 'b00005lzly' and x[0][1] == 'http://www.google.com/
base/feeds/snippets/13823221823254120257').collect()
print(similarityTest)
[(('b00005lzly', 'http://www.google.com/base/feeds/snippets/13823221823254120257'),
4.286548413995203e-06)]
```

Part 5: Analysis

Now we have an authoritative list of record-pair similarities, but we need a way to use those similarities to decide if two records are duplicates or not. The simplest approach is to pick a **threshold**. Pairs whose similarity is above the threshold are declared duplicates, and pairs below the threshold are declared distinct.

To decide where to set the threshold we need to understand what kind of errors result at different levels. If we set the threshold too low, we get more **false positives**, that is, record-pairs we say are duplicates that in reality are not. If we set the threshold too high, we get more **false negatives**, that is, record-pairs that really are duplicates but that we miss.

ER algorithms are evaluated by the common metrics of information retrieval and search called **precision** and **recall**. Precision asks of all the record-pairs marked duplicates, what fraction are true duplicates? Recall asks of all the true duplicates in the data, what fraction did we successfully find? As with false positives and false negatives, there is a trade-off between precision and recall. A third metric, called **F-measure**, takes the harmonic mean of precision and recall to measure overall goodness in a single value:

$$Fmeasure = 2 * ((precision * recall) / (precision + recall))$$

In this part, we use the "gold standard" mapping from the included file to look up true duplicates, and the results of Part 4.

(5a) Counting True Positives, False Positives, and False Negatives

We need functions that count True Positives (true duplicates above the threshold), and False Positives and False Negatives:

- We start with creating the `simsFullRDD` from our `similaritiesFullRDD` that consists of a pair of ((Amazon ID, Google URL), simlarity score).
- From this RDD, we create an RDD consisting of only the similarity scores.
- To look up the similarity scores for true duplicates, we perform a left outer join using the `goldStandard` RDD and extract the similarities scores.

```
>
# Create an RDD of ((Amazon ID, Google URL), similarity score)
simsFullRDD = similaritiesFullRDD.map(lambda x: ("%s %s" % (x[0][0], x[0][1]), x[1]))
assert (simsFullRDD.count() == 2441100)

# Create an RDD of just the similarity scores
simsFullValuesRDD = (simsFullRDD
                     .map(lambda x: x[1])
                     .cache())
assert (simsFullValuesRDD.count() == 2441100)

# Look up all similarity scores for true duplicates

# This helper function will return the similarity score for records that are in the gold standard and the
simsFullRDD (True positives), and will return 0 for records that are in the gold standard but not in simsFullRDD
(False Negatives).
def gs_value(record):
    if (record[1][1] is None):
        return 0
    else:
        return record[1][1]

# Join the gold standard and simsFullRDD, and then extract the similarities scores using the helper function
trueDupSimsRDD = (goldStandard
                  .leftOuterJoin(simsFullRDD)
                  .map(gs_value)
                  .cache())
print('There are %s true duplicates.' % trueDupSimsRDD.count())
There are 1300 true duplicates.
```

```
assert(trueDupSimsRDD.count() == 1300)
```

The next step is to pick a threshold between 0 and 1 for the count of True Positives (true duplicates above the threshold). However, we would like to explore many different thresholds.

To do this, we divide the space of thresholds into 100 bins, and take the following actions:

- We use Spark Accumulators to implement our counting function. We define a custom accumulator type, `VectorAccumulatorParam`, along with functions to initialize the accumulator's vector to zero, and to add two vectors. Note that we have to use the += operator because you can only add to an accumulator.
- We create a helper function to create a list with one entry (bit) set to a value and all others set to 0.
- We create 101 bins for the 100 threshold values between 0 and 1.
- Now, for each similarity score, we can compute the false positives. We do this by adding each similarity score to the appropriate bin of the vector. Then we remove true positives from the vector by using the gold standard data.
- We define functions for computing false positive and negative and true positives, for a given threshold.

```
>
from pyspark.accumulators import AccumulatorParam
class VectorAccumulatorParam(AccumulatorParam):
    # Initialize the VectorAccumulator to 0
    def zero(self, value):
        return [0] * len(value)

    # Add two VectorAccumulator variables
    def addInPlace(self, val1, val2):
        for i in range(len(val1)):
            val1[i] += val2[i]
        return val1

# Return a list with entry x set to value and all other entries set to 0
def set_bit(x, value, length):
    bits = []
    for y in range(length):
        if (x == y):
            bits.append(value)
        else:
            bits.append(0)
    return bits

# Pre-bin counts of false positives for different threshold ranges
BINS = 101
nthresholds = 100
def bin(similarity):
    return int(similarity * nthresholds)

# fpCounts[i] = number of entries (possible false positives) where bin(similarity) == i
zeros = [0] * BINS
fpCounts = sc.accumulator(zeros, VectorAccumulatorParam())

def add_element(score):
    global fpCounts
    b = bin(score)
```

```
    fpCounts += set_bit(b, 1, BINS)

simsFullValuesRDD.foreach(add_element)

# Remove true positives from FP counts
def sub_element(score):
    global fpCounts
    b = bin(score)
    fpCounts += set_bit(b, -1, BINS)

trueDupSimsRDD.foreach(sub_element)

def falsepos(threshold):
    fpList = fpCounts.value
    return sum([fpList[b] for b in range(0, BINS) if float(b) / nthresholds >= threshold])

def falseneg(threshold):
    return trueDupSimsRDD.filter(lambda x: x < threshold).count()

def truepos(threshold):
    return trueDupSimsRDD.count() - falsenegDict[threshold]
```

(5b) Precision, Recall, and F-measures

We define functions so that we can compute the Precision, Recall, and F-measure as a function of threshold value:

- Precision = true-positives / (true-positives + false-positives).
- Recall = true-positives / (true-positives + false-negatives).
- F-measure = 2 * (Recall * Precision) / (Recall + Precision).

```
>
# Precision = true-positives / (true-positives + false-positives)
# Recall = true-positives / (true-positives + false-negatives)
# F-measure = 2 x Recall x Precision / (Recall + Precision)

def precision(threshold):
    tp = trueposDict[threshold]
    return float(tp) / (tp + falseposDict[threshold])

def recall(threshold):
    tp = trueposDict[threshold]
    return float(tp) / (tp + falsenegDict[threshold])

def fmeasure(threshold):
    r = recall(threshold)
    p = precision(threshold)
    return 2 * r * p / (r + p)
```

(5c) Line Plots

We can make line plots of precision, recall, and F-measure as a function of threshold value, for thresholds between 0.0 and 1.0.

```
>
thresholds = [float(n) / nthresholds for n in range(0, nthresholds)]
falseposDict = dict([(t, falsepos(t)) for t in thresholds])
```

```
falsenegDict = dict([(t, falseneg(t)) for t in thresholds])
trueposDict = dict([(t, truepos(t)) for t in thresholds])

precisions = [precision(t) for t in thresholds]
recalls = [recall(t) for t in thresholds]
fmeasures = [fmeasure(t) for t in thresholds]

print(precisions[0], fmeasures[0])
0.0005325468026709271 0.0010645266950540452

assert (abs(precisions[0] - 0.000532546802671) < 0.0000001)
assert (abs(fmeasures[0] - 0.00106452669505) < 0.0000001)


fig = plt.figure()
plt.plot(thresholds, precisions)
plt.plot(thresholds, recalls)
plt.plot(thresholds, fmeasures)
plt.legend(['Precision', 'Recall', 'F-measure'])
display(fig)
pass


# Create a DataFrame and visualize using display()
graph = [(t, precision(t), recall(t),fmeasure(t)) for t in thresholds]
graphRDD = sc.parallelize(graph)

graphRow = graphRDD.map(lambda txyz: Row(threshold=txyz[0], precision=txyz[1], recall=txyz[2],
fmeasure=txyz[3]))
graphDF = sqlContext.createDataFrame(graphRow)
display(graphDF)
```

(5d) Discussion

State-of-the-art tools can get an F-measure of about 60% on this dataset. In this assignment, our best F-measure is closer to 40%. Look at some examples of errors (both False Positives and False Negatives) and think about what went wrong. There are several ways we might improve our simple classifier, including:

- Using additional attributes
- Performing better featurization of our textual data (e.g., stemming, n-grams, etc.)
- Using different similarity functions