Text Analysis and Entity Resolution

Entity resolution is a common, yet difficult problem in data cleaning and integration. This assignment will demonstrate how we can use Apache Spark to apply powerful and scalable text analysis techniques and perform entity resolution across two datasets of commercial products.

Entity Resolution, or "Record linkage" is the term used by statisticians, epidemiologists, and historians, among others, to describe the process of joining records from one data source with another that describe the same entity. Our terms with the same meaning include, "entity disambiguation/linking", duplicate detection", "deduplication", "record matching", "(reference) reconciliation", "object identification", "data/information integration", and “conflation".

Entity Resolution (ER) refers to the task of finding records in a dataset that refer to the same entity across different data sources (e.g., data files, books, websites, databases). ER is necessary when joining datasets based on entities that may or may not share a common identifier (e.g., database key, URI, National identification number), as may be the case due to differences in record shape, storage location, and/or curator style or preference. A dataset that has undergone ER may be referred to as being cross-linked.

Part 0: Preliminaries

We read in each of the files and create an RDD consisting of lines. For each of the data files ("Google.csv", "Amazon.csv", and their small samples), we want to parse the IDs out of each record. The IDs are the first column of the file (they are URLs for Google, and alphanumeric strings for Amazon). Omitting the headers, we load these data files into pair RDDs where the *mapping ID* is the key, and the value is a string consisting of the name/title, description, and manufacturer from the record.

The file format of an Amazon line is:

```
"id","title","description","manufacturer","price"
```

The file format of a Google line is:

```
"id","name","description","manufacturer","price"
```

```
>
import re
DATAFILE_PATTERN = '^(.+),"(.+)",(.*),(.*),(.*)'

def removeQuotes(s):
    """ Remove quotation marks from an input string
    Args:
        s (str): input string that might have the quote "" characters
    Returns:
        str: a string without the quote characters
    """
    return ''.join(i for i in s if i!='"')


def parseDatafileLine(datafileLine):
    """ Parse a line of the data file using the specified regular expression pattern
    Args:
```

```
    datafileLine (str): input string that is a line from the data file
  Returns:
    str: a string parsed using the given regular expression and without the quote characters
  """
  match = re.search(DATAFILE_PATTERN, datafileLine)
  if match is None:
    print 'Invalid datafile line: %s' % datafileLine
    return (datafileLine, -1)
  elif match.group(1) == '"id"':
    print 'Header datafile line: %s' % datafileLine
    return (datafileLine, 0)
  else:
    product = '%s %s %s' % (match.group(2), match.group(3), match.group(4))
    return ((removeQuotes(match.group(1)), product), 1)
```

Now let's create an RDD consisting of lines for each of the files and map them into pair RDDs. To this end, we will use a data set originally from Google's metric-learning project. Download the zip file for this assignment and uncompress it in your Spark directory, which has total six files as follows:

- Google.csv, the Google Products dataset
- Amazon.csv, the Amazon dataset
- Google_small.csv, 200 records sampled from the Google data
- Amazon_small.csv, 200 records sampled from the Amazon data
- Amazon_Google_perfectMapping.csv, the "gold standard" mapping for algorithm evaluation that contains all the true mappings between entities in two datasets (i.e., describe the same entity in the real world)
- stopwords.txt, a list of common English words

>

```
GOOGLE_PATH = 'Google.csv'
GOOGLE_SMALL_PATH = 'Google_small.csv'
AMAZON_PATH = 'Amazon.csv'
AMAZON_SMALL_PATH = 'Amazon_small.csv'
GOLD_STANDARD_PATH = 'Amazon_Google_perfectMapping.csv'
STOPWORDS_PATH = 'stopwords.txt'

def parseData(filename):
  """ Parse a data file
  Args:
    filename (str): input file name of the data file
  Returns:
    RDD: a RDD of parsed lines
  """
  return (sc
        .textFile(filename, 4, 0)
        .map(parseDatafileLine))

def loadData(filename):
  """ Load a data file
  Args:
    path (str): input file name of the data file
  Returns:
    RDD: a RDD of parsed valid lines
  """
  raw = parseData(filename).cache()
  failed = (raw
```

```
            .filter(lambda s: s[1] == -1)
            .map(lambda s: s[0]))
    for line in failed.take(10):
        print '%s - Invalid datafile line: %s' % (filename, line)
    valid = (raw
            .filter(lambda s: s[1] == 1)
            .map(lambda s: s[0])
            .cache())
    print '%s - Read %d lines, successfully parsed %d lines, failed to parse %d lines' % (filename,
                                                                raw.count(),
                                                                valid.count(),
                                                                failed.count())
    assert failed.count() == 0
    assert raw.count() == (valid.count() + 1)
    return valid

googleSmall = loadData(GOOGLE_SMALL_PATH)
google = loadData(GOOGLE_PATH)
amazonSmall = loadData(AMAZON_SMALL_PATH)
amazon = loadData(AMAZON_PATH)
```

```
Google_small.csv - Read 201 lines, successfully parsed 200 lines, failed to parse 0 lines
Google.csv - Read 3227 lines, successfully parsed 3226 lines, failed to parse 0 lines
Amazon_small.csv - Read 201 lines, successfully parsed 200 lines, failed to parse 0 lines
Amazon.csv - Read 1364 lines, successfully parsed 1363 lines, failed to parse 0 lines
```

Let's examine the lines that were just loaded in the two subset (small) files - one from Google and one from Amazon.

```
>
for line in googleSmall.take(3):
    print 'google: %s: %s\n' % (line[0], line[1])

for line in amazonSmall.take(3):
    print 'amazon: %s: %s\n' % (line[0], line[1])
```

```
google: http://www.google.com/base/feeds/snippets/11448761432933644608: spanish vocabulary builder
"expand your vocabulary! contains fun lessons that both teach and entertain you'll quickly find yourself mastering
new terms. includes games and more!"

google: http://www.google.com/base/feeds/snippets/8175198959985911471: topics presents: museums of world
"5 cd-rom set. step behind the velvet rope to examine some of the most treasured collections of antiquities art
and inventions. includes the following the louvre - virtual visit 25 rooms in full screen interactive video detailed
map of the louvre ..."

google: http://www.google.com/base/feeds/snippets/18445827127704822533: sierrahome hse hallmark card
studio special edition win 98 me 2000 xp "hallmark card studio special edition (win 98 me 2000 xp)" "sierrahome"

amazon: b000jz4hqo: clickart 950 000 - premier image pack (dvd-rom)  "broderbund"

amazon: b0006zf55o: ca international - arcserve lap/desktop oem 30pk "oem arcserve backup v11.1 win 30u for
laptops and desktops" "computer associates"

amazon: b00004tkvy: noah's ark activity center (jewel case ages 3-8)  "victory multimedia"
```

Part 1: ER as Text Similarity - Bags of Words

A simple approach to entity resolution is to treat all records as strings and compute their similarity with a string distance function. In this part, we will build some components for performing bag-of-words text-analysis, and then use them to compute record similarity. Bag-of-words is a conceptually simple yet powerful approach to text analysis.

The idea is to treat strings, a.k.a. documents, as unordered collections of words, or tokens, i.e., as bags of words. A "token" is the result of parsing the document down to the elements we consider "atomic" for the task at hand. Tokens can be things like words, numbers, acronyms, or other exotica like word-roots or fixed-length character strings. Bag of words techniques all apply to any sort of token, so when we say "bag-of-words" we really mean "bag-of-tokens," strictly speaking. Tokens become the atomic unit of text comparison. If we want to compare two documents, we count how many tokens they share in common. If we want to search for documents with keyword queries (this is what Google does), then we turn the keywords into tokens and find documents that contain them. The power of this approach is that it makes string comparisons insensitive to small differences that probably do not affect meaning much, for example, punctuation and word order.

(1a) Tokenize a String
Implement the function `simpleTokenize(string)` that takes a string and returns a list of non-empty tokens in the string. `simpleTokenize` should split strings using the provided regular expression. Since we want to make token-matching case insensitive, make sure all tokens are turned lower-case. Give an interpretation, in natural language, of what the regular expression, `split_regex`, matches. If you need help with Regular Expressions, try the site regex101 where you can interactively explore the results of applying different regular expressions to strings. *Note that \W includes the "_" character*. You should use re.split() to perform the string split. Also, make sure you remove any empty tokens.

>
```
# TODO: Replace <FILL IN> with appropriate code
quickbrownfox = 'A quick brown fox jumps over the lazy dog.'
split_regex = r'\W+'

def simpleTokenize(string):
    """ A simple implementation of input string tokenization
    Args:
        string (str): input string
    Returns:
        list: a list of tokens
    """
    return <FILL IN>

print simpleTokenize(quickbrownfox)
['a', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', ‘dog']

print simpleTokenize(' ')
[]

print simpleTokenize('!!!!123A/456_B/789C.123A')
['123a', '456_b', '789c', '123a']

print simpleTokenize('fox fox')
['fox', 'fox']
```

(1b) Removing stopwords
Stopwords are common (English) words that do not contribute much to the content or meaning of a document (e.g., "the", "a", "is", "to", etc.). Stopwords add noise to bag-of-words comparisons, so they are usually excluded. Using the included file "stopwords.txt", implement `tokenize`, an improved tokenizer that does not emit stopwords.

>
```
# TODO: Replace <FILL IN> with appropriate code
stopfile = STOPWORDS_PATH
stopwords = set(sc.textFile(stopfile).collect())

print 'These are the stopwords: %s' % stopwords
```
These are the stopwords: set([u'all', u'just', u'being', u'over', u'both', u'through', u'yourselves', u'its', u'before', u'with', u'had', u'should', u'to', u'only', u'under', u'ours', u'has', u'do', u'them', u'his', u'very', u'they', u'not', u'during', u'now', u'him', u'nor', u'did', u'these', u't', u'each', u'where', u'because', u'doing', u'theirs', u'some', u'are', u'our', u'ourselves', u'out', u'what', u'for', u'below', u'does', u'above', u'between', u'she', u'be', u'we', u'after', u'here', u'hers', u'by', u'on', u'about', u'of', u'against', u's', u'or', u'own', u'into', u'yourself', u'down', u'your', u'from', u'her', u'whom', u'there', u'been', u'few', u'too', u'themselves', u'was', u'until', u'more', u'himself', u'that', u'but', u'off', u'herself', u'than', u'those', u'he', u'me', u'myself', u'this', u'up', u'will', u'while', u'can', u'were', u'my', u'and', u'then', u'is', u'in', u'am', u'it', u'an', u'as', u'itself', u'at', u'have', u'further', u'their', u'if', u'again', u'no', u'when', u'same', u'any', u'how', u'other', u'which', u'you', u'who', u'most', u'such', u'why', u'a', u'don', u'i', u'having', u'so', u'the', u'yours', u'once'])

```
def tokenize(string):
    """ An implementation of input string tokenization that excludes stopwords
    Args:
        string (str): input string
    Returns:
        list: a list of tokens without stopwords
    """
    return <FILL IN>

print tokenize(quickbrownfox)
```
['quick', 'brown', 'fox', 'jumps', 'lazy', 'dog']

```
print tokenize("Why a the?")
```
[]

```
print tokenize("Being at the_?")
```
['the_']

(1c) Tokenizing the Small Datasets
Now let's tokenize the two *small* datasets. For each ID in a dataset, `tokenize` the values, and then count the total number of tokens. How many tokens, total, are there in the two datasets?

>
```
# TODO: Replace <FILL IN> with appropriate code
amazonRecToToken = amazonSmall.<FILL IN>
googleRecToToken = googleSmall.<FILL IN>

def countTokens(vendorRDD):
    """ Count and return the number of tokens
```

```
    Args:
        vendorRDD (RDD of (recordId, tokenizedValue)): Pair tuple of record ID to tokenized output
    Returns:
        count: count of all tokens
    """
    return <FILL IN>

totalTokens = countTokens(amazonRecToToken) + countTokens(googleRecToToken)
print 'There are %s tokens in the combined datasets' % totalTokens
```
```
There are 22520 tokens in the combined datasets
```

(1d) Amazon Record with the Most Tokens
Which Amazon record has the biggest number of tokens? In other words, you want to sort the records and get the one with the largest count of tokens.

```
>
# TODO: Replace <FILL IN> with appropriate code
def findBiggestRecord(vendorRDD):
    """ Find and return the record with the largest number of tokens
    Args:
        vendorRDD (RDD of (recordId, tokens)): input Pair Tuple of record ID and tokens
    Returns:
        list: a list of 1 Pair Tuple of record ID and tokens
    """
    return <FILL IN>

biggestRecordAmazon = findBiggestRecord(amazonRecToToken)
print 'The Amazon record with ID "%s" has the most tokens (%s)' % (biggestRecordAmazon[0][0],
                                                                  len(biggestRecordAmazon[0][1]))
```
```
The Amazon record with ID "b000o24l3q" has the most tokens (1547)
```

Part 2: ER as Text Similarity - Weighted Bag-of-Words using TF-IDF

Bag-of-words comparisons are not very good when all tokens are treated the same: some tokens are more important than others. Weights give us a way to specify which tokens to favor. With weights, when we compare documents, instead of counting common tokens, we sum up the weights of common tokens. A good heuristic for assigning weights is called "Term-Frequency/Inverse-Document-Frequency," or TF-IDF for short.

TF
TF rewards tokens that appear many times in the same document. It is computed as the frequency of a token in a document, that is, if document *d* contains 100 tokens and token *t* appears in *d* 5 times, then the TF weight of *t* in *d* is *5/100 = 1/20*. The intuition for TF is that if a word occurs often in a document, then it is more important to the meaning of the document.

IDF
IDF rewards tokens that are rare overall in a dataset. The intuition is that it is more significant if two documents share a rare word than a common one. IDF weight for a token, *t*, in a set of documents, *U*, and is computed as follows:
- Let *N* be the total number of documents in *U*.
- Find *n(t)*, the number of documents in *U* that contain *t*.

• Then *IDF(t) = N/n(t)*.
Note that *n(t)/N* is the frequency of *t* in *U*, and *N/n(t)* is the inverse frequency.

Sometimes token weights depend on the document the token belongs to, that is, the same token may have a different weight when it's found in different documents. We call these weights *local* weights. TF is an example of a local weight, because it depends on the length of the source. On the other hand, some token weights only depend on the token, and are the same everywhere that token is found. We call these weights *global*, and IDF is one such weight.

TF-IDF
Finally, to bring it all together, the total TF-IDF weight for a token in a document is the product of its TF and IDF weights.

(2a) Implement a TF Function
Implement `tf(tokens)` that takes a list of tokens and returns a Python dictionary mapping tokens to TF weights.

The steps your function should perform are:
• Create an empty Python dictionary.
• For each of the tokens in the input `tokens` list, count 1 for each occurrence and add the token to the dictionary.
• For each of the tokens in the dictionary, divide the token's count by the total number of tokens in the input `tokens` list.

>
```
# TODO: Replace <FILL IN> with appropriate code
def tf(tokens):
    """ Compute TF
    Args:
        tokens (list of str): input list of tokens from tokenize
    Returns:
        dictionary: a dictionary of tokens to its TF values
    """
    <FILL IN>
    return <FILL IN>

print tf(tokenize(quickbrownfox))
{'brown': 0.16666666666666666, 'lazy': 0.16666666666666666, 'jumps': 0.16666666666666666, 'fox':
0.16666666666666666, 'dog': 0.16666666666666666, 'quick': 0.16666666666666666}

print tf(tokenize('one_ one_ two!'))
{'two': 0.3333333333333333, 'one_': 0.6666666666666666}
```

(2b) Create a Corpus
Create a pair RDD called `corpusRDD`, consisting of a combination of the two small datasets, `amazonRecToToken` and `googleRecToToken`. Each element of the `corpusRDD` should be a pair consisting of a key from one of the small datasets (ID or URL) and the value is the associated value for that key from the small datasets.

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

print corpusRDD.count()
```

(2c) Implement an IDFs Function
Implement `idfs` that assigns an IDF weight to every unique token in an RDD called `corpus`. The function should return a pair RDD where the `key` is the unique token and value is the IDF weight for the token.

Recall that the IDF weight for a token, *t*, in a set of documents, *U*, is computed as follows:
- Let *N* be the total number of documents in *U*.
- Find *n(t)*, the number of documents in *U* that contain *t*.
- Then *IDF(t) = N/n(t)*.

The steps your function should perform are:
- Calculate *N*. Think about how you can calculate *N* from the input RDD.
- Create an RDD (not a pair RDD) containing the unique tokens from each document in the input `corpus`. For each document, you should only include a token once, *even if it appears multiple times in that document*. (Hint: Consider using Python Sets, which automatically removes duplicate entries.)
- For each of the unique tokens, count how many times it appears in the document and then compute the IDF for that token: *N/n(t)*.

Use your `idfs` to compute the IDF weights for all tokens in `corpusRDD` (the combined small datasets). How many unique tokens are there?

```
>
# TODO: Replace <FILL IN> with appropriate code
def idfs(corpus):
    """ Compute IDF
    Args:
        corpus (RDD): input corpus
    Returns:
        RDD: a RDD of (token, IDF value)
    """
    N = <FILL IN>
    uniqueTokens = corpus.<FILL IN>
    tokenCountPairTuple = uniqueTokens.<FILL IN>
    tokenSumPairTuple = tokenCountPairTuple.<FILL IN>
    return (tokenSumPairTuple.<FILL IN>)

idfsSmall = idfs(amazonRecToToken.union(googleRecToToken))
uniqueTokenCount = idfsSmall.count()

print 'There are %s unique tokens in the small datasets.' % uniqueTokenCount
There are 4772 unique tokens in the small datasets.

tokenSmallestIdf = idfsSmall.takeOrdered(1, lambda s: s[1])[0]
print tokenSmallestIdf[0]
software

print abs(tokenSmallestIdf[1] - 4.25531914894) < 0.0000000001
True
```

(2d) Tokens with the Smallest IDF
Print out the 11 tokens with the smallest IDF in the combined small dataset.

>
```
smallIDFTokens = idfsSmall.takeOrdered(11, lambda s: s[1])
print smallIDFTokens
```
[('software', 4.25531914893617), ('new', 6.896551724137931), ('features', 6.896551724137931), ('use', 7.017543859649122), ('complete', 7.2727272727272725), ('easy', 7.6923076923076925), ('create', 8.333333333333334), ('system', 8.333333333333334), ('cd', 8.333333333333334), ('1', 8.51063829787234), ('windows', 8.51063829787234)]

(2e) IDF Histogram

Plot a histogram of IDF values. Be sure to use appropriate scaling and bucketing for the data. First plot the histogram using `matplotlib`.

>
```
import matplotlib.pyplot as plt

small_idf_values = idfsSmall.map(lambda s: s[1]).collect()
fig = plt.figure(figsize=(8,3))
plt.hist(small_idf_values, 50, log=True)
display(fig)
pass

from pyspark.sql import Row

# Create a DataFrame and visualize using display()
idfsToCountRow = idfsSmall.map(lambda (x, y): Row(token=x, value=y))
idfsToCountDF = sqlContext.createDataFrame(idfsToCountRow)
display(idfsToCountDF)
```

(2f) Implement a TF-IDF Function

Use your `tf` function to implement a `tfidf(tokens, idfs)` function that takes a list of tokens from a document and a Python dictionary of IDF weights and returns a Python dictionary mapping individual tokens to total TF-IDF weights.

The steps your function should perform are:
- Calculate the token frequencies (TF) for `tokens`.
- Create a Python dictionary where each token maps to the token's frequency times the token's IDF weight.

Use your `tfidf` function to compute the weights of Amazon product record 'b000hkgj8k'. To do this, we need to extract the record for the token from the tokenized small Amazon dataset and we need to convert the IDFs for the small dataset into a Python dictionary. We can do the first part, by using a `filter()` transformation to extract the matching record and a `collect()` action to return the value to the driver.

For the second part, we use the collectAsMap() action to return the IDFs to the driver as a Python dictionary.

>
```
# TODO: Replace <FILL IN> with appropriate code
def tfidf(tokens, idfs):
    """ Compute TF-IDF
    Args:
        tokens (list of str): input list of tokens from tokenize
```

```
    idfs (dictionary): record to IDF value
  Returns:
    dictionary: a dictionary of records to TF-IDF values
  """
  tfs = <FILL IN>
  tfIdfDict = <FILL IN>
  return tfIdfDict

recb000hkgj8k = amazonRecToToken.filter(lambda x: x[0] == 'b000hkgj8k').collect()[0][1]
idfsSmallWeights = idfsSmall.collectAsMap()
rec_b000hkgj8k_weights = tfidf(recb000hkgj8k, idfsSmallWeights)

print 'Amazon record "b000hkgj8k" has tokens and weights:\n%s' % rec_b000hkgj8k_weights
Amazon record "b000hkgj8k" has tokens and weights:
{'autocad': 33.33333333333333, 'autodesk': 8.333333333333332, 'courseware': 66.66666666666666, 'psg':
33.33333333333333, '2007': 3.5087719298245617, 'customizing': 16.666666666666664, 'interface':
3.0303030303030303}
```

Part 3: ER as Text Similarity - Cosine Similarity

Now we are ready to do text comparisons in a formal way. The metric of string distance we will use is called cosine similarity. We will treat each document as a vector in some high dimensional space. Then, to compare two documents we compute the cosine of the angle between their two document vectors. This is *much* easier than it sounds.

The first question to answer is how do we represent documents as vectors? The answer is familiar: bag-of-words! We treat each unique token as a dimension, and treat token weights as magnitudes in their respective token dimensions. For example, suppose we use simple counts as weights, and we want to interpret the string "Hello, world! Goodbye, world!" as a vector. Then in the "hello" and "goodbye" dimensions the vector has value 1, in the "world" dimension it has value 2, and it is zero in all other dimensions.

The next question is: given two vectors how do we find the cosine of the angle between them? Recall the formula for the dot product of two vectors:

$$a \cdot b = |a||b|\cos\theta$$

Here $a \cdot b = \sum a_i b_i$ is the ordinary dot product of two vectors, and $|a| = \text{sqrt}(\sum a_i^2)$ is the norm of a.

We can rearrange terms and solve for the cosine to find it is simply the normalized dot product of the vectors. With our vector model, the dot product and norm computations are simple functions of the bag-of-words document representations, so we now have a formal way to compute similarity:

$$similarity = \cos\theta = a \cdot b \,/\, (|a||b|) = \sum a_i b_i \,/\, (\text{sqrt}(\sum a_i^2)\text{sqrt}(\sum b_i^2))$$

Setting aside the algebra, the geometric interpretation is more intuitive. The angle between two document vectors is small if they share many tokens in common, because they are pointing in roughly the same direction. For that case, the cosine of the angle will be large. Otherwise, if the angle is large (and they have few words in common), the cosine is small. Therefore, cosine similarity scales proportionally with our intuitive sense of similarity.

(3a) Implement the Components of a `cosineSimilarity` Function
Implement the components of a `cosineSimilarity` function. Use the `tokenize` and `tfidf` functions, and the IDF weights from Part 2 for extracting tokens and assigning them weights.

The steps you should perform are:

• Define a function `dotprod` that takes two Python dictionaries and produces the dot product of them, where the dot product is defined as the sum of the product of values for tokens that appear in *both* dictionaries.
• Define a function `norm` that returns the square root of the dot product of a dictionary and itself.
• Define a function `cossim` that returns the dot product of two dictionaries divided by the norm of the first dictionary and then by the norm of the second dictionary.

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

def dotprod(a, b):
    """ Compute dot product
    Args:
        a (dictionary): first dictionary of record to value
        b (dictionary): second dictionary of record to value
    Returns:
        dotProd: result of the dot product with the two input dictionaries
    """
    return <FILL IN>

def norm(a):
    """ Compute square root of the dot product
    Args:
        a (dictionary): a dictionary of record to value
    Returns:
        norm (float): the square root of the dot product value
    """
    return <FILL IN>

def cossim(a, b):
    """ Compute cosine similarity
    Args:
        a (dictionary): first dictionary of record to value
        b (dictionary): second dictionary of record to value
    Returns:
        cossim: dot product of two dictionaries divided by the norm of the first dictionary and
                then by the norm of the second dictionary
    """
    return <FILL IN>

testVec1 = {'foo': 2, 'bar': 3, 'baz': 5 }
testVec2 = {'foo': 1, 'bar': 0, 'baz': 20 }
dp = dotprod(testVec1, testVec2)
nm = norm(testVec1)
print dp, nm
102 6.16441400297
```

(3b) Implement a `cosineSimilarity` Function

Implement a `cosineSimilarity(string1, string2, idfsDictionary)` function that takes two strings and a dictionary of IDF weights, and computes their cosine similarity in the context of some global IDF weights.

The steps you should perform are:

• Apply your `tfidf` function to the tokenized first and second strings, using the dictionary of IDF weights.
• Compute and return your `cossim` function applied to the results of the two `tfidf` functions.

>
```
# TODO: Replace <FILL IN> with appropriate code
def cosineSimilarity(string1, string2, idfsDictionary):
    """ Compute cosine similarity between two strings
    Args:
        string1 (str): first string
        string2 (str): second string
        idfsDictionary (dictionary): a dictionary of IDF values
    Returns:
        cossim: cosine similarity value
    """
    w1 = tfidf(<FILL IN>)
    w2 = tfidf(<FILL IN>)
    return cossim(w1, w2)

cossimAdobe = cosineSimilarity('Adobe Photoshop',
                               'Adobe Illustrator',
                               idfsSmallWeights)

print cossimAdobe
0.0577243382163
```

(3c) Perform Entity Resolution
Now we can finally do some entity resolution! For every product record in the small Google dataset, use your `cosineSimilarity` function to compute its similarity to every record in the small Amazon dataset. Then, build a dictionary mapping `(Google URL, Amazon ID)` tuples to similarity scores between 0 and 1. We'll do this computation two different ways, first we'll do it without a broadcast variable, and then we'll use a broadcast variable.

The steps you should perform are:
• Create an RDD that is a combination of the small Google and small Amazon datasets that has as elements all pairs of elements (a, b) where a is in self and b is in other, which is called cross/cartesian product. The result will be an RDD of the form: `[ ((Google URL1, Google String1), (Amazon ID1, Amazon String1)), ((Google URL1, Google String1), (Amazon ID2, Amazon String2)), ((Google URL2, Google String2), (Amazon ID1, Amazon String1)), ... ]`
• Define a worker function that given an element from the combination RDD computes the cosineSimlarity for the two records in the element.
• Apply the worker function to every element in the RDD.

Now, compute the similarity between Amazon record `b000o24l3q` and Google record `http://www.google.com/base/feeds/snippets/17242822440574356561`.

>
```
# TODO: Replace <FILL IN> with appropriate code
crossSmall = (googleSmall
              .<FILL IN>
              .cache())

def computeSimilarity(record):
```

```
    """ Compute similarity on a combination record
    Args:
        record: a pair, (google record, amazon record)
    Returns:
        3-tuple: (google URL, amazon ID, cosine similarity value)
    """
    googleRec = record[0]
    amazonRec = record[1]
    googleURL = <FILL IN>
    amazonID = <FILL IN>
    googleValue = <FILL IN>
    amazonValue = <FILL IN>
    cs = cosineSimilarity(<FILL IN>, idfsSmallWeights)
    return (googleURL, amazonID, cs)

similarities = (crossSmall
                .<FILL IN>
                .cache())

def similar(amazonID, googleURL):
    """ Return similarity value
    Args:
        amazonID: amazon ID
        googleURL: google URL
    Returns:
        similar: cosine similarity value
    """
    return (similarities
            .filter(lambda record: (record[0] == googleURL and record[1] == amazonID))
            .collect()[0][2])

similarityAmazonGoogle = similar('b000o24l3q', 'http://www.google.com/base/feeds/snippets/
17242822440574356561')
print 'Requested similarity is %s.' % similarityAmazonGoogle
Requested similarity is 0.000303171940451.
```

(3d) Perform Entity Resolution with Broadcast Variables

The solution in (3c) works well for small datasets, but it requires Spark to (automatically) send the `idfsSmallWeights` variable to all the workers. If we didn't `cache()` similarities, then it might have to be recreated if we run `similar()` multiple times. This would cause Spark to send `idfsSmallWeights` every time.

Instead, we can use a broadcast variable - we define the broadcast variable in the driver and then we can refer to it in each worker. Spark saves the broadcast variable at each worker, so it is only sent once.

The steps you should perform are:

- Define a `computeSimilarityBroadcast` function that given an element from the combination RDD computes the cosine simlarity for the two records in the element. This will be the same as the worker function `computeSimilarity` in (3c) except that it uses a broadcast variable.
- Apply the worker function to every element in the RDD.

Again, compute the similarity between Amazon record `b000o24l3q` and Google record `http://www.google.com/base/feeds/snippets/17242822440574356561`.

```
>
# TODO: Replace <FILL IN> with appropriate code
def computeSimilarityBroadcast(record):
    """ Compute similarity on a combination record, using Broadcast variable
    Args:
        record: a pair, (google record, amazon record)
    Returns:
        pair: a pair, (google URL, amazon ID, cosine similarity value)
    """
    googleRec = record[0]
    amazonRec = record[1]
    googleURL = <FILL IN>
    amazonID = <FILL IN>
    googleValue = <FILL IN>
    amazonValue = <FILL IN>
    cs = cosineSimilarity(<FILL IN>, idfsSmallBroadcast.value)
    return (googleURL, amazonID, cs)

idfsSmallBroadcast = sc.broadcast(idfsSmallWeights)
similaritiesBroadcast = (crossSmall
                         .<FILL IN>
                         .cache())

def similarBroadcast(amazonID, googleURL):
    """ Return similarity value, computed using Broadcast variable
    Args:
        amazonID: amazon ID
        googleURL: google URL
    Returns:
        similar: cosine similarity value
    """
    return (similaritiesBroadcast
            .filter(lambda record: (record[0] == googleURL and record[1] == amazonID))
            .collect()[0][2])

similarityAmazonGoogleBroadcast = similarBroadcast('b000o24l3q', 'http://www.google.com/base/feeds/
snippets/17242822440574356561')
print 'Requested similarity is %s.' % similarityAmazonGoogleBroadcast
Requested similarity is 0.000303171940451.

print len(idfsSmallBroadcast.value)
4772
```

(3e) Perform a Gold Standard Evaluation

First, we'll load the "gold standard" data and use it to answer several questions. We read and parse the Gold Standard data, where the format of each line is "Amazon Product ID","Google URL". The resulting RDD has elements of the form ("AmazonID GoogleURL", 'gold').

```
>
GOLDFILE_PATTERN = '^(.+),(.+)'

# Parse each line of a data file using the specified regular expression pattern
def parse_goldfile_line(goldfile_line):
    """ Parse a line from the 'golden standard' data file
```

```
    Args:
        goldfile_line: a line of data
    Returns:
        pair: ((key, 'gold', 1 if successful or else 0))
    """
    match = re.search(GOLDFILE_PATTERN, goldfile_line)
    if match is None:
        print 'Invalid goldfile line: %s' % goldfile_line
        return (goldfile_line, -1)
    elif match.group(1) == '"idAmazon"':
        print 'Header datafile line: %s' % goldfile_line
        return (goldfile_line, 0)
    else:
        key = '%s %s' % (removeQuotes(match.group(1)), removeQuotes(match.group(2)))
        return ((key, 'gold'), 1)

goldfile = GOLD_STANDARD_PATH
gsRaw = (sc
         .textFile(goldfile)
         .map(parse_goldfile_line)
         .cache())

gsFailed = (gsRaw
            .filter(lambda s: s[1] == -1)
            .map(lambda s: s[0]))
for line in gsFailed.take(10):
    print 'Invalid goldfile line: %s' % line

goldStandard = (gsRaw
                .filter(lambda s: s[1] == 1)
                .map(lambda s: s[0])
                .cache())

print 'Read %d lines, successfully parsed %d lines, failed to parse %d lines' % (gsRaw.count(),
                                                                                 goldStandard.count(),
                                                                                 gsFailed.count())
Read 1301 lines, successfully parsed 1300 lines, failed to parse 0 lines

assert (gsFailed.count() == 0)
assert (gsRaw.count() == (goldStandard.count() + 1))
```

Using the “gold standard” data we can answer the following questions:

- How many true duplicate pairs are there in the small datasets?
- What is the average similarity score for true duplicates?
- What about for non-duplicates? The steps you should perform are:
    - Create a new `sims` RDD from the `similaritiesBroadcast` RDD, where each element consists of a pair of the form ("AmazonID GoogleURL", cosineSimilarityScore). An example entry from `sims` is: ('`b000bi7uqs http://www.google.com/base/feeds/snippets/18403148885652932189`', 0.40202896125621296).
    - Combine the `sims` RDD with the `goldStandard` RDD by creating a new `trueDupsRDD` RDD that has the just the cosine similarity scores for those "AmazonID GoogleURL" pairs that appear in both the `sims` RDD and `goldStandard` RDD. (Hint: you can do this using the join() transformation.)
    - Count the number of true duplicate pairs in the `trueDupsRDD` dataset.

• Compute the average similarity score for true duplicates in the `trueDupsRDD` datasets. Remember to use `float` for calculation.

• Create a new `nonDupsRDD` RDD that has the just the cosine similarity scores for those "AmazonID GoogleURL" pairs from the `similaritiesBroadcast` RDD that *do not* appear in both the `sims` RDD and `goldStandard` RDD. (Hint: you can do this using the leftOuterJoin() transformation.)

• Compute the average similarity score for non-duplicates in the last datasets. Remember to use `float` for calculation.

>

```
# TODO: Replace <FILL IN> with appropriate code
sims = similaritiesBroadcast.<FILL IN>)

trueDupsRDD = (sims
               .<FILL IN>)
trueDupsCount = trueDupsRDD.<FILL IN>
avgSimDups = <FILL IN>

nonDupsRDD = (sims
              .<FILL IN>)
avgSimNon = <FILL IN>

print 'There are %s true duplicates.' % trueDupsCount
There are 146 true duplicates.

print 'The average similarity of true duplicates is %s.' % avgSimDups
The average similarity of true duplicates is 0.264332573435.

print 'And for non duplicates, it is %s.' % avgSimNon
And for non duplicates, it is 0.00123476304656.
```