Entity resolution

An application of the techniques for detecting similarities among documents is that of delete duplicate records referring to a same entity when merging two or more data sources. This kind of processing is typically referred to as *entity resolution*, or *record linkage*, although other more or less equivalent terms in technical jargon exist (such as *entity disambiguation*, *entity linking*, *duplicate detection*, *deduplication*, *record matching*, *conflation*, *reconciliation*, or generically *data integration*)

Thus the term entity resolution refers to the detection of different records in a database all referring to a same entity, although not equal one another, typically after an integration process involving different data sources has taken place.

Tools

This tutorial is based on the use of basic features of Python 2.7, Spark 1.4.1 coupled with the pySpark API and with the matplotlib graphical library. Some use of the standard libraries for dealing with regular expressions is expected: if needed, check the Web site regex101 (https://regex101.com/), allowing to interactively apply regular expressions to strings.

Files

The used data are from the metric-learning (https://code.google.com/archive/p/metric-learning/source) project. Precisely, we will use the following files, stored in the `data` directory:

- `Google.csv`, product dataset from Google,
- `Amazon.csv`, product dataset from da Amazon,
- `Google_small.csv`, 200 records samapled from `Google.csv`,
- `Amazon_small.csv` 200 records sampled from `Amazon.csv`,
- `Amazon_Google_perfectMapping.csv`, mapping between records in the two datasets referring to a same product,
- `stopwords.txt`, a list of stop words in English.

In the first part of the tutorial we will use the sampled files, in order to be able to conduct interactive experimentations. The file showing pairs of records describing a same product (information typically referred to as*gold standard* o *ground truth*) will be used in order to evaluate the proposed algorithm performances.

We start by storing the names of these files in some constants:

In [111]:

```
import sys
import os

base_dir = os.path.join('data')

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'
```

0. Preliminary operations

We start by creating a RDD for each dataset. The content of these files is line bases, each line referring to a product, described using the format

    `"id","name","description","producer","price"`

(the first line of each file is a header, thus it will be discarded). It is worth noting that the double quotes are actually contained in the various strings, and that ids have a different form in the two datasets: URLs for Google, and alphanumeric codes for Amazon.

Starting from the raw data, we build RDDs whose elements are pairs having as key the id and as value the remaining part of each line. Special care needs to be considered when parsing double quotes, because missing data (for instance when considering the producer) correspond to an empty line (without double quotes) delimited by lines. Thus we will use commas as references in order to build a regexp for each dataset line (five fields separated by commas, among which the first two are always specified):

In [112]:

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

Once checked out that a line matches this regular expression we can remove double quotes, when needed, using the following function:

In [113]:

```
def remove_quotes(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!='"')
```

We can now write a function to properly parse a dataset line, checking that the latter satisfies the required format (otherwise returning a pair containing the line and the special value `-1`) and properly identifying the first line of the file, containing headers (returning in this case a pair containing the line and the special value `0`). Finally, the values for each field will be extracted in order to build a pair having as key the product id (discarding double quotes), and as value the remaining fields concatenated. Such object will be returned as first element of a pair ending with the special value `1`. Checking the second element of each pair it will bepossible to verify how many lines in a file have been correctly converted.

In [114]:

```
def parse_datafile_line(datafile_line):
    """ 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, datafile_line)
    if match is None:
        print 'Invalid datafile line: %s' % datafile_line
        return (datafile_line, -1)
    elif match.group(1) == '"id"':
        print 'Header datafile line: %s' % datafile_line
        return (datafile_line, 0)
    else:
        product = '%s %s %s' % (match.group(2), match.group(3), match.group(4)
        return ((remove_quotes(match.group(1)), product), 1)
```

The next step consists in defining a function returning a RDD containing an element for each line of a data file, and in mapping `parse_dataline_file` to this RDD, *caching the result*.

In [115]:

```
def parse_data(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(parse_datafile_line)
            )
```

It is now possible to write a function which, given a dataset as parameter, builds the corresponding RDD and verifies that no conversion error occurred. That is obtained as follows:

- building an RDD filtering the number of pairs containing the special values `-1` and selecting their first element (having as result the lines causing conversion errors);
- printing the content of (at most) ten such lines;
- building the RDD corresponding to the paris containing the value `1` and selecting their first element (that is, a pair `(id, product)` obtained after a valid conversion);
- caching the converted values RDD;

- verifying that no conversino errors occurred and that all the file lines have been converted;
- returning the RDD containing the converted values.

In [116]:

```
def load_data(path):
    """ Load a data file

    Args:
        path (str): input file name of the data file

    Returns:
        RDD: a RDD of parsed valid lines
    """

    filename = os.path.join(base_dir, path)
    raw = parse_data(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' % (path, 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') % (path,
                                          raw.count(),
                                          valid.count(),
                                          failed.count())
    assert failed.count() == 0
    assert raw.count() == (valid.count() + 1)
    return valid
```

Using these functions it is now possible to convert the datasets which will be used henceforth.

In [117]:

```
google_small = load_data(GOOGLE_SMALL_PATH)
google = load_data(GOOGLE_PATH)
amazon_small = load_data(AMAZON_SMALL_PATH)
amazon = load_data(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, fai
led 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, fai
led to parse 0 lines
```

Let's check the content of some records in the converted datasets:

In [118]:

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

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

```
google: http://www.google.com/base/feeds/snippets/114487614329336
44608: (http://www.google.com/base/feeds/snippets/114487614329336
44608:) "spanish vocabulary builder" "expand your vocabulary! con
tains fun lessons that both teach and entertain you'll quickly fi
nd yourself mastering new terms. includes games and more!"

google: http://www.google.com/base/feeds/snippets/817519895998591
1471: (http://www.google.com/base/feeds/snippets/8175198959985911
471:) "topics presents: museums of world" "5 cd-rom set. step beh
ind the velvet rope to examine some of the most treasured collect
ions of antiquities art and inventions. includes the following th
e louvre - virtual visit 25 rooms in full screen interactive vide
o detailed map of the louvre ..."

google: http://www.google.com/base/feeds/snippets/184458271277048
22533: (http://www.google.com/base/feeds/snippets/184458271277048
22533:) "sierrahome hse hallmark card studio special edition win
98 me 2000 xp" "hallmark card studio special edition (win 98 me 2
000 xp)" "sierrahome"

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

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

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

1. Encoding documents using *bag of words*

A natural way to perform entity resolution on textual documents is that of considering each record as a string and using a suitable distance in order to compute the similarity between two documents. Precisely we will use the so-called *bag of words* approach, in which a textual documents is encoded as a set of words, or even better of *tokens* occurring in it.

> **Note**: the term *token* is more appropriate if we want to emphasize that we are not necessarily working with words, rather with indivisible «atoms» either identified with words or also with fixed-length substrings (k-gram) occurring in the text.

Tokens will be elements on the basis of element comparison, whose similarity will be expressed in terms of the number of common tokens.

1.1 Extract tokens from a string

We implement a function `simple_tokenize` taking as argument a string and returning the list of its occurring tokens. Here tokens will be words separated by whitespace. This function suitable uses regular expressions and will be case insensitive, and also avoid considering empty tokens.

In [119]:

```
quickbrownfox = 'A quick brown fox jumps over the lazy dog.'
split_regex = r'\W+'

def simple_tokenize(string):
    """ A simple implementation of input string tokenization
    Args:
        string (str): input string
    Returns:
        list: a list of tokens
    """
    return [s for s in re.split(split_regex, string.lower()) if s != '']
```

Let's check that the function correctly works on a simple test case, that it does not consider whitespace and that it does not eliminate duplicate tokens.

In [120]:

```
print simple_tokenize(quickbrownfox)

assert(simple_tokenize(quickbrownfox) ==
       ['a','quick','brown','fox','jumps','over','the','lazy','dog'])
assert(simple_tokenize(' ') == [])
assert(simple_tokenize('!!!!123A/456_B/789C.123A') == ['123a','456_b','789c','
assert(simple_tokenize('fox fox') == ['fox', 'fox'])
```

```
['a', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'd
og']
```

1.2 Removing stop words

The term *stop word* denote words commonly occurring in texts written in a given language, thus not conveying specific information. This is why it is advisable to remove such words before extract tokens from a document. Using the file `stopwords.txt` we can implement the function `tokenize`, analogous to `simple_tokenize` but also ignoring stop words.

In [121]:

```
stopfile = os.path.join(base_dir, STOPWORDS_PATH)
stopwords = set(sc.textFile(stopfile).collect())

print 'These are the stopwords: %s' % stopwords

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 [s for s in re.split(split_regex, string.lower())
            if s != '' and not s in 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'd
o', 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'beca
use', 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'your
self', u'down', u'your', u'from', u'her', u'whom', u'there', u'be
en', 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'thos
e', 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'i
t', u'an', u'as', u'itself', u'at', u'have', u'further', u'thei
r', u'if', u'again', u'no', u'when', u'same', u'any', u'how', u'o
ther', 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'])
```

Also in this case, we verify the correct behaviour through simple tests: the function should eliminate stop words yet consider all remaining terms.

In [122]:

```
print tokenize(quickbrownfox)

assert(tokenize("Why a the?") == [])
assert(tokenize("Being at the_?") == ['the_'])
assert(tokenize(quickbrownfox) == ['quick','brown','fox','jumps','lazy','dog'])
```

```
['quick', 'brown', 'fox', 'jumps', 'lazy', 'dog']
```

1.3 Extract tokens from the sampled datasets

We can extract tokens from the sampled datasets through repeated application of `tokenize` to all elements of the corresponding RDD.

In [123]:

```
amazon_rec_to_token = amazon_small.map(lambda s: (s[0], tokenize(s[1])))
google_rec_to_token = google_small.map(lambda s: (s[0], tokenize(s[1])))
```

As already done before, we check the result counting the number of tokens in the two datasets.

In [124]:

```
def count_tokens(vendorRDD):
    """ Count and return the number of tokens
    Args:
        vendorRDD (RDD of (recordId, tokenizedValue)): Pair tuple of record ID
    Returns:
        count: count of all tokens
    """
    return vendorRDD.map(lambda s: len(s[1])).reduce(lambda a, b: a+b)
```

In [125]:

```
total_tokens = count_tokens(amazon_rec_to_token) + count_tokens(google_rec_to_
print 'There are %s tokens in the combined datasets' % total_tokens
assert(total_tokens == 22520)
```

```
There are 22520 tokens in the combined datasets
```

1.4 Finding out which document has more tokens

Which is the document in the Amazon dataset having the highest number of tokens? In order to find it out it is sufficient to suitably sort the records.

In [126]:

```
def find_biggest_record(vendorRDD):
    """ Find and return the record with the largest number of tokens

    Args:
        vendorRDD (RDD of (recordId, tokens)): input Pair Tuple of record ID a

    Returns:
        list: a list of 1 Pair Tuple of record ID and tokens
    """
    return vendorRDD.takeOrdered(1, key = lambda s: -1*len(s[1]))

biggest_record_amazon = find_biggest_record(amazon_rec_to_token)
print 'The Amazon record with ID "%s" has the most tokens (%s)' % (biggest_rec
                                                                  len(biggest

assert(biggest_record_amazon[0][0] == 'b000o24l3q')
assert(len(biggest_record_amazon[0][1]) == 1547)
```

```
The Amazon record with ID "b000o24l3q" has the most tokens (1547)
```

2. Use of the TF.IDF measure

To improve the technique based on the bag of words approach it is important to assign different weights to the tokens in a document, in order to reflect the most relevant terms. An heuristic helping to find such terms is the *TF.IDF measure* (Term Frequency times Inverse Document Frequency), obtained through multiplication of the two indicators defined hereafter.

Term Frequency

The *Term Frequency* index rewards tokens occurring several times in a same document. It is computed as the relative frequency of a token in a document. In other words, if document d containg 100 tokens and among the latter t occurs five times, the Term Frequency of t in d is $\mathrm{TF}(t, d) = \frac{5}{100} = \frac{1}{20}$.

Inverse Document Frequency

The *Inverse Document Frequency* rewards tokens occurring frequently in a dataset. For a given token t and a corpus of documents U, the Inverse Document Frequency of t equals $\mathrm{IDF}(t) = \frac{N}{n(t)}$, where N denotes the size of U and $n(t)$ is the number of documents in U containing t.

TF.IDF

The TF.IDF measure of a token t in a document d equals the product between Term Frequency and Inverse Document Frequency: $\mathrm{TF.IDF}(t, d) = \mathrm{TF}(t, d) \cdot \mathrm{IDF(t)}$. Informally speaking, a high TF.IDF corresponds to a token with several occurrences in some documents yet rare in most documents. Thus, when computing the similarity between two documents, a higher weight should be given to the fact that both contain a same term with high TF.IDF measure.

2.1 Implement a Term Frequency function

We can implement a `tf` function which, given a list of token, returns a dictionary mapping each token in the corresponding TF value. The function will:

- build an empty dictionary;
- scan the list, adding each token to the dictionary as key associated to 1 if such key does not already occurs, otherwise increments by 1 the value associated to that key;
- divides value associated to each key in the dictionary by the total number of tokens in the list.

In [127]:

```
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
    """
    val = {}
    for t in tokens:
        if t in val:
            val[t] += 1
        else:
            val[t] = 1.0
    for t in val:
        val[t] /= len(tokens)
    return val
```

Also in this case, we verify the `tf` function using some basic tests.

In [128]:

```
print tf(tokenize(quickbrownfox))
tf_test = tf(tokenize(quickbrownfox))
assert(tf_test == {'brown': 0.16666666666666666, 'lazy': 0.16666666666666666,
                           'jumps': 0.16666666666666666, 'fox': 0.1666666666
                           'dog': 0.16666666666666666, 'quick': 0.1666666666
tf_test2 = tf(tokenize('one_ one_ two!'))
assert(tf_test2 == {'one_': 0.6666666666666666, 'two': 0.3333333333333333})
```

```
{'brown': 0.16666666666666666, 'lazy': 0.16666666666666666, 'jump
s': 0.16666666666666666, 'fox': 0.16666666666666666, 'dog': 0.166
66666666666666, 'quick': 0.16666666666666666}
```

2.2 Build a document corpus

The next step consists in creating a RDD acting as *corpus* containing all documents to be analyzed, thus being the result of a merge operation between the Amazon and the Google datasets. Each element of this new RDD will be a pair whose key and value are respectively the id and the description of a product (after the cleaning process).

In [129]:

```
corpusRDD = google_rec_to_token.union(amazon_rec_to_token)
```

In [130]:

```
assert(corpusRDD.count() == 400)
```

2.3 Implement an IDF function

The implementation of a function `idfs` assigning the IDF value to each token occurring in the document corpus should return a pair RDD whose pairs (t, i) have a token as key and its IDF measure as value. This function will:

- compute N, the number of documents in the corpus;
- create a RDD containing an element for each document in the corpus, where the element is a list containing each *unique* token in the document; in other words, a token in a document should be included only once, *even when there are several occurrences of that token in the document*;
- for each unique token t, count its number of occurrences $n(t)$ in the corpus and successively compute the IDF value.

In [131]:

```
def idfs(corpus):
    """ Compute IDF

    Args:
        corpus (RDD): input corpus

    Returns:
        RDD: a RDD of (record ID, IDF value)
    """

    N = corpus.count() * 1.0
    unique_tokens = corpus.map(lambda s: list(set(s[1])))
    token_count_pair_tuple = unique_tokens.flatMap(lambda t: [(s, 1) for s in
    token_sum_pair_tuple = token_count_pair_tuple.reduceByKey(lambda a, b: a+b
    return token_sum_pair_tuple.map(lambda s: (s[0], N/s[1]))
```

Using the `idfs` function it is now possible to compute the IDF values for all tokens in `corpusRDD`, the union of the two sampled datasets, and find out how many unique tokens are there.

In [132]:

```
idfs_small = idfs(corpusRDD)
unique_token_count = idfs_small.count()

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

assert(unique_token_count == 4772)
token_smallest_IDF = idfs_small.takeOrdered(1, lambda s: s[1])[0]
assert(token_smallest_IDF[0] == 'software')
assert(abs(token_smallest_IDF[1] - 4.25531914894) < 0.0000000001)
```

```
There are 4772 unique tokens in the small datasets.
```

2.4 Find the tokens with smaller IDF value

Through a suitable sorting process of the RDD containing the IDF values it is possible to print the eleven tokens having smallest IDF value.

In [134]:

```
small_IDF_tokens = idfs_small.takeOrdered(11, lambda s: s[1])
print small_IDF_tokens
```

```
[('software', 4.25531914893617), ('new', 6.896551724137931), ('fe
atures', 6.896551724137931), ('use', 7.017543859649122), ('comple
te', 7.2727272727272725), ('easy', 7.6923076923076925), ('creat
e', 8.333333333333334), ('system', 8.333333333333334), ('cd', 8.3
33333333333334), ('1', 8.51063829787234), ('windows', 8.510638297
87234)]
```

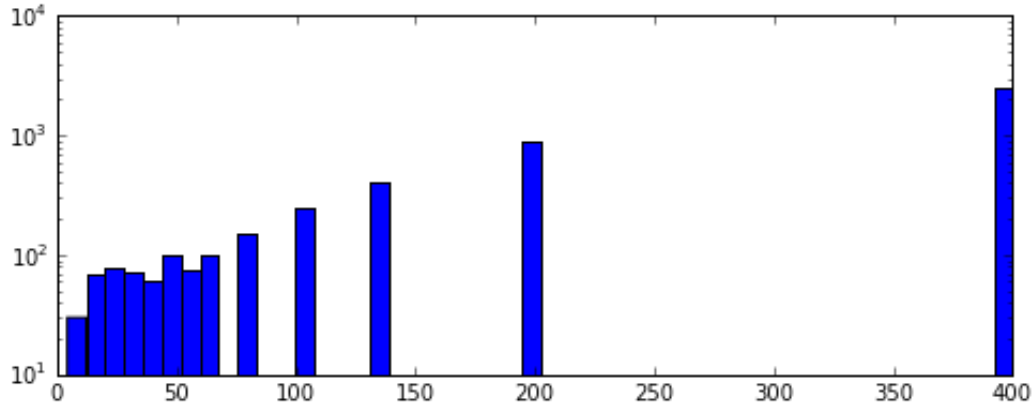
2.5 Histogram of the IDF values

We can now show an histogram of the IDF values in the corpus, using the `matplotlib` library.

In [135]:

```
%matplotlib inline
import matplotlib.pyplot as plt

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



2.6 Implement a TF.IDF function

We can use the `tf` function in order to implement a new `tfidf` function which, given as arguments a lis of token and the dictionary returned by `idfs`, returns in turn a dictionary mapping each token in the list to the corresponding TF.IDF value.

The `tfidf` function will

- compute the TF values for all tokens in the list, and
- build a dictionary associating each token to the product of the corresponding TF and IDF values.

In [136]:

```
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 = tf(tokens)
    tf_idf_dict = {t: tfs[t]*idfs[t] for t in tokens}
    return tf_idf_dict
```

We apply the `tfidf` function in order to compute the TF.IDF measure for the Amazon product whose ID is `b000hkgj8k`. This will require to extract the corresponding record from the dataset, as well as to convert the RDD containing the IFS values for the corpus into a dictionary. The first task is easily accomplished through filtering, and the second one can be carried out calling the `collectAsMap` function.

In [137]:

```
recb000hkgj8k = amazon_rec_to_token.filter(lambda x: x[0] == 'b000hkgj8k').col
idfs_small_weights = idfs_small.collectAsMap()
rec_b000hkgj8k_weights = tfidf(recb000hkgj8k, idfs_small_weights)

print 'Amazon record "b000hkgj8k" has tokens and weights:\n%s' % rec_b000hkgj8

assert(rec_b000hkgj8k_weights ==
                   {'autocad': 33.33333333333333, 'autodesk': 8.33333333333333
                    'courseware': 66.66666666666666, 'psg': 33.33333333333333,
                    '2007': 3.5087719298245617, 'customizing': 16.666666666666
                    'interface': 3.0303030303030303})
```

```
Amazon record "b000hkgj8k" has tokens and weights:
{'autocad': 33.33333333333333, 'autodesk': 8.333333333333332, 'co
urseware': 66.66666666666666, 'psg': 33.33333333333333, '2007':
3.5087719298245617, 'customizing': 16.666666666666664, 'interfac
e': 3.0303030303030303}
```

3. Use of the cosine distance

One of the possible distances between documents is the so-called *cosine distance*, interpreting two objects as directions in a space and computing the cosine of the angle between these directions.

The first difficulty consists in encoding documents as directions in a space, thus as vectors. Actually, starting from the tokenization of documents, we can figure out a space having a dimension for each possible token in the corpus: a generic document will have as component in the dimension corresponding to a token the corresponding TF.IDF value (with the obvious extension of nullifying the components for dimensions whose tokens do not occur in a document.

The second problem is that of evaluate the length of an angle between vectors: this can be easily obtained noting that $a \cdot b = ||a||||b|| \cos\theta$, where $a \cdot b$ denotes the inner product between two vectors a and b, θ denotes the angle between the same vectors and $||a||$ denotes the norm a. Therefore

$$\text{sim}(a, b) = \cos\theta = \frac{a \cdot b}{||a||||b||} = \frac{\sum a_i b_i}{\sqrt{\sum a_i^2}\sqrt{\sum b_i^2}} \quad (*)$$

.

It should be noted that, although the considered space has in principle a huge number of dimensions, each vector can be encoded as a dictionary containing only the non-null components, thus exploiting the sparsity properties. More precisely, for each token t having TF.IDF measure $i \neq 0$, in that dictionary the key t will be associated to the value i.

3.1 Implement dot product and norm for sparse vectors

Using the `tokenize` and `tfidf` functions and exploiting the sparse representation in dictionaries it is possible to define:

- a `dotprod` function aving as arguments the dictionaries describing two sparse vectors and returning the corresponding dot product (computed as the sum of products of values corresponding to keys occurring in *both* dictionaries);
- a `norm` function returning the norm of a sparse vector a passed as a dictionary (computed as the square root of the dot product between a and a);
- a `cossinm` function returning the cosine distance between two sparse vectors passed as arguments, exploiting the equation (*) and the functions defined in previous points.

In [139]:

```
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 sum([a[t] * b[t] for t in a if t in b])

def norm(a):
    """ Compute square root of the dot product

    Args:
        a (dictionary): a dictionary of record to value

    Returns:
        norm: a dictionary of tokens to its TF values
    """
    return math.sqrt(dotprod(a, a))

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 fir
                then by the norm of the second dictionary
    """
    return dotprod(a, b)/(norm(a) * norm(b))
```

Also in this case, we can check the correct behaviour of these functions using simple test cases.

In [141]:

```
testVec1 = {'foo': 2, 'bar': 3, 'baz': 5 }
testVec2 = {'foo': 1, 'bar': 0, 'baz': 20 }
dp = dotprod(testVec1, testVec2)
nm = norm(testVec1)
print dp, nm
assert(dp == 102)
assert(abs(nm - 6.16441400297) < 0.0000001)
```

```
102 6.16441400297
```

In [142]:

```
cossim(testVec1, testVec2)
```

Out[142]:

```
0.8262970212292282
```

3.2 Implementare una funzione per il calcolo della similarità

Possiamo ora implementare una funzione `cosine_similarity` che, date due stringhe che descrivono documenti e un dizionario per i valori IDF in un corpus, calcola e restituisce la distanza del coseno tra i corrispondenti vettori nello spazio dei token. I passi necessari sono:

- tokenizzare le due stringhe e successivamente utilizzarle unitamente al dizionario per invocare la funzione `tfidf`, e
- invocare la funzione `cossim` e restituirne il risultato.

In [143]:

```
def cosine_similarity(string1, string2, idfs_dictionary):
    """ 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(tokenize(string1), idfs_dictionary)
    w2 = tfidf(tokenize(string2), idfs_dictionary)
    return cossim(w1, w2)
```

È quindi ora possibile verificare la similarità tra due prodottti, date le loro descrizioni:

In [144]:

```
cossimAdobe = cosine_similarity('Adobe Photoshop',
                                'Adobe Illustrator',
                                idfs_small_weights)

print cossimAdobe
assert(abs(cossimAdobe - 0.0577243382163) < 0.0000001)
```

```
0.0577243382163
```

3.3 Entitiy resolution procedure

All tools we built allow us to carry out the entity resolution procedure, consisting in considering each product in the Google dataset and computing its cosine distance w.r.t. all products in the Amazon datasta. Precisely we will proceed as follows:

- we will create a RDD containing all pairs of products in the two datasets, expressed as a tuple `((Google URL, Google String), (Amazon ID, Amazon String))`;
- we will define a function accepting as argument such a tuple as argument and returning the cosine distance between the two products thereing contained;
- we will finally apply that function to all elements in the RDD created in the first point.

In [146]:

```
cross_small = (google_small
               .cartesian(amazon_small)
               .cache())

def compute_similarity(record):
    """ Compute similarity on a combination record

    Args:
        record: a pair, (google record, amazon record)

    Returns:
        pair: a pair, (google URL, amazon ID, cosine similarity value)
    """

    google_rec = record[0]
    amazon_rec = record[1]
    google_URL = google_rec[0]
    amazon_ID = amazon_rec[0]
    google_value = google_rec[1]
    amazon_value = amazon_rec[1]

    cs = cosine_similarity(google_value, amazon_value, idfs_small_weights)
    return (google_URL, amazon_ID, cs)

similarities = (cross_small
                .map(lambda r: compute_similarity(r))
                .cache())

def similar(amazon_ID, google_URL):
    """ Return similarity value

    Args:
        amazon_ID: amazon ID
        google_URL: google URL

    Returns:
        similar: cosine similarity value
    """

    return (similarities
            .filter(lambda record: (record[0] == google_URL and record[1] == a
            .collect()[0][2])
```

We can now compute and show, for instance, the similarity measure between the Amazon product having ID `b000o24l3q` and the Google product having URL `http://www.google.com/base/feeds/snippets/17242822440574356561`.

In [147]:

```
similarity_Amazon_Google = similar('b000o24l3q', 'http://www.google.com/base/f
print 'Requested similarity is %s.' % similarity_Amazon_Google
assert(abs(similarity_Amazon_Google - 0.000303171940451) < 0.0000001)
```

```
Requested similarity is 0.000303171940451.
```

The obtained similarity degree is low. We can check that the corresponding documents are actually different, printing the first 300 characters of both:

In [37]:

```
' '.join(amazon_rec_to_token.filter(lambda p: p[0]=='b000o24l3q').take(1)[0][1
```

Out[37]:

```
'adobe premiere pro cs3 upgrade note upgrade version adobe premie
re pro cs3 tell story maximum impact using adobe premiere pro cs3
upgrade software start finish solution efficient video production
includes adobe encore cs3 adobe onlocation cs3 windows formerly a
ward winning dv rack hd save time set c'
```

In [38]:

```
' '.join(google_rec_to_token.filter(lambda p: p[0]=='http://www.google.com/bas
```

Out[38]:

```
'diana ross supremes yamaha best diana ross supremes smart pianos
oft innovative software series enables disklavier mark iii piano
perform world popular cds using yamaha pianosmart technology comp
anion diskette magically empower disklavier mark iii accompany'
```

Actually the document contents look different: one of them deals probably wito software, the other one with music. Things are different for the `b00004tkvy` and `http://www.google.com/base/feeds/snippets/18441110047404795849` documents:

In [39]:

```
similarity_Amazon_Google = similar('b00004tkvy', 'http://www.google.com/base/f
print 'Requested similarity is %s.' % similarity_Amazon_Google
```

```
Requested similarity is 0.733049938579.
```

Now there is a high similarity, and indeed the documents have several common terms:

In [40]:

```
' '.join(amazon_rec_to_token.filter(lambda p: p[0]=='b00004tkvy').take(1)[0][1
```

Out[40]:

```
'noah ark activity center jewel case ages 3 8 victory multimedia'
```

In [41]:

```
' '.join(google_rec_to_token.filter(lambda p: p[0]=='http://www.google.com/bas
```

Out[41]:

```
'beginners bible noah ark activity center activity center'
```