Transforming documents into a sparse matrix

In this activity, we will learn one way to transform documents from text to a sparse matrix that can be used for different data mining tasks.

In [5]:

```python
import numpy as np
import scipy as sp
%matplotlib inline
import matplotlib.pyplot as plt
from collections import defaultdict
```

In [6]:

```python
# open docs file and read its lines
with open("data/docs.txt", "r", encoding="utf8") as fh:
    lines = fh.readlines()
```

How many documents do we have? Write some code to print the number of lines in docs.txt.

In [7]:

```python
len(lines)
```

Out[7]:

```
60
```

Create a list variable called `docs` that contains a list of lists, one for each document, s.t. the ith list is a list of all lower-cased words in the ith document. Print out the total number of words in the collection and the average number of words per document.

In [8]:

```python
# transform docs into lists of words
docs = [l.split() for l in lines]
```

The functions plotWf and plotDf below compute and plot the word frequency distribution (how many times each word is found in the collection) and document frequency distributions (how many documents each word is found in), respectively. Note how they are constructed. Then, execute the cell below to register the functions. In the following cell, execute the functions to plot the frequency distributions.

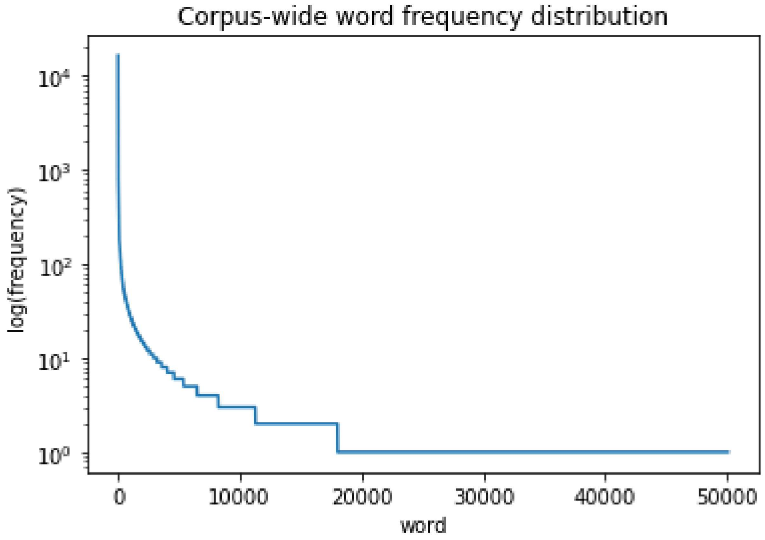
In [9]:

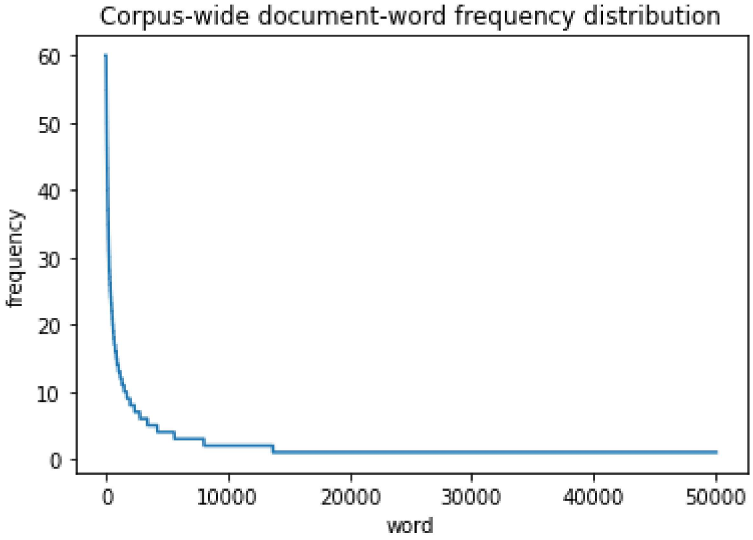
```
def plotWf(docs, plot=True, logscale=True):
    r"""Get collection-wide word frequencies and optionally plot them."""
    words = defaultdict(int)
    for d in docs:
        for w in d:
            words[w] += 1
    if plot is True:
        plt.plot(sorted(words.values(), reverse=True))
        plt.xlabel('word')
        plt.ylabel('frequency')
        if logscale is True:
            plt.yscale('log')
            plt.ylabel('log(frequency)')
        plt.title('Corpus-wide word frequency distribution')
        plt.show()
    return words

def plotDf(docs, plot=True, logscale=False):
    r"""Get collection-wide word frequencies and optionally plot them."""
    # document word frequency
    df = defaultdict(int)
    for d in docs:
        for w in set(d):
            df[w] += 1
    if plot is True:
        plt.plot(sorted(df.values(), reverse=True))
        plt.xlabel('word')
        plt.ylabel('frequency')
        if logscale is True:
            plt.yscale('log')
            plt.ylabel('log(frequency)')
        plt.title('Corpus-wide document-word frequency distribution')
        plt.show()
    return df
```

In [10]:

```
_ = plotWf(docs)
_ = plotDf(docs)
```





The `filterLen` function filters out words that may be too short based on the minlen parameter. Execute the code below to see the difference between a document with all words and a document with 3-letter and shorter words removed.

In [11]:

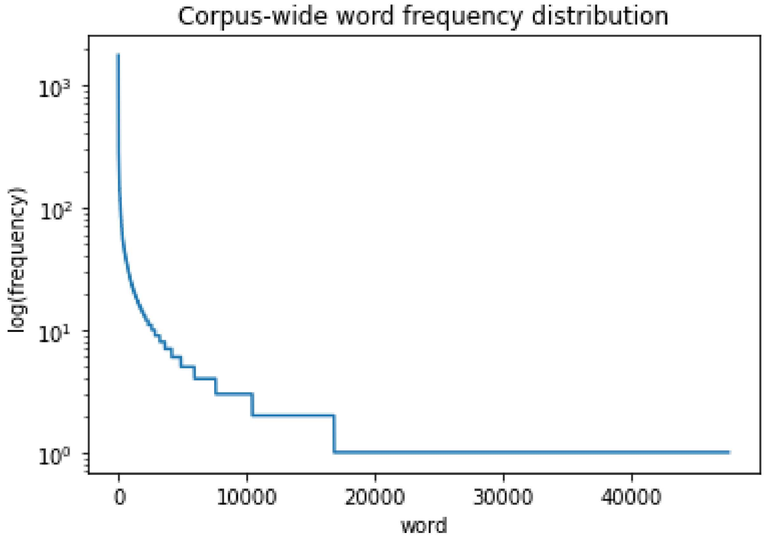
```
def filterLen(docs, minlen):
    r""" filter out terms that are too short.
    docs is a list of lists, each inner list is a document represented as a l
    minlen is the minimum length of the word to keep
    """
    return [ [t for t in d if len(t) >= minlen ] for d in docs ]
docs1 = filterLen(docs, 4)
print(len(docs[0]), docs[0][:20])
print(len(docs1[0]), docs1[0][:20])
```

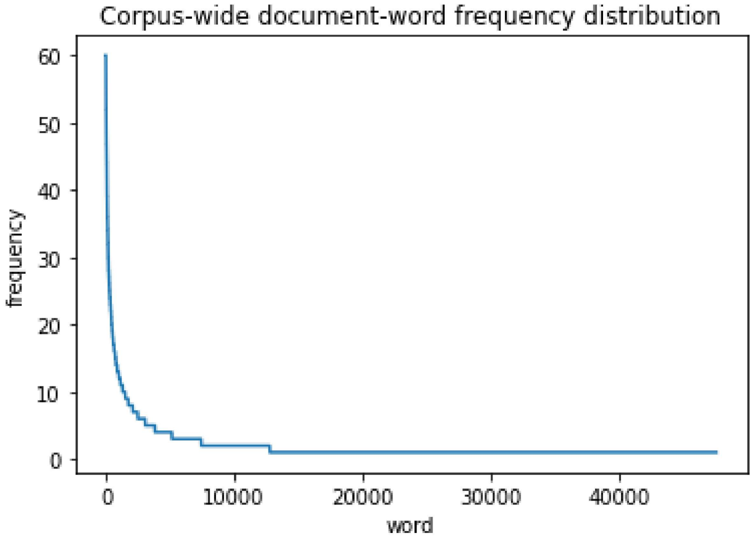
```
3813 ['Octopus', 'The', 'octopus', 'is', 'a', 'cephalopod', 'of', 'the', 'o
rder', 'Octopoda.', 'Octopuses', 'have', 'two', 'eyes', 'and', 'four', 'pai
rs', 'of', 'arms', 'and']
2555 ['Octopus', 'octopus', 'cephalopod', 'order', 'Octopoda.', 'Octopuse
s', 'have', 'eyes', 'four', 'pairs', 'arms', 'like', 'other', 'cephalopod
s', 'bilaterally', 'symmetric.', 'octopus', 'hard', 'beak,', 'with']
```

Re-execute the `plotWf` and `plotDf` functions to see the difference after filering.

In [12]:

```
_ = plotWf(docs1)
_ = plotDf(docs1)
```





The `build_matrix` function will transform a collection represented as a list of lists of words into a sparse matrix. The `csr_info` function will display some statistics about the sparse matrix. Study the functions and then run them for the two document collections, as follows:

> - mat = build_matrix(docs)
> - mat1 = build_matrix(docs1)

Finally, print out matrix stats for the two matrices:

> - csr_info(mat, "mat", non_empy=True)
> - csr_info(mat1, "mat1", non_empy=True)

Make sure you run the cell below first in order to register the functions.

In [13]:

```
from collections import Counter
from scipy.sparse import csr_matrix
def build_matrix(docs):
    r""" Build sparse matrix from a list of documents,
    each of which is a list of word/terms in the document.
    """
    nrows = len(docs)
    idx = {}
    tid = 0
    nnz = 0
    for d in docs:
        nnz += len(set(d))
        for w in d:
            if w not in idx:
                idx[w] = tid
                tid += 1
    ncols = len(idx)

    # set up memory
    ind = np.zeros(nnz, dtype=np.int)
    val = np.zeros(nnz, dtype=np.double)
    ptr = np.zeros(nrows+1, dtype=np.int)
    i = 0  # document ID / row counter
    n = 0  # non-zero counter
    # transfer values
    for d in docs:
        cnt = Counter(d)
        keys = list(k for k,_ in cnt.most_common())
        l = len(keys)
        for j,k in enumerate(keys):
            ind[j+n] = idx[k]
            val[j+n] = cnt[k]
        ptr[i+1] = ptr[i] + l
        n += l
        i += 1

    mat = csr_matrix((val, ind, ptr), shape=(nrows, ncols), dtype=np.double)
    mat.sort_indices()

    return mat


def csr_info(mat, name="", non_empy=False):
    r""" Print out info about this CSR matrix. If non_empy,
    report number of non-empty rows and cols as well
    """
    if non_empy:
        print("%s [nrows %d (%d non-empty), ncols %d (%d non-empty), nnz %d]"
                name, mat.shape[0],
                sum(1 if mat.indptr[i+1] > mat.indptr[i] else 0
                for i in range(mat.shape[0])),
                mat.shape[1], len(np.unique(mat.indices)),
                len(mat.data)))
    else:
        print( "%s [nrows %d, ncols %d, nnz %d]" % (name,
                mat.shape[0], mat.shape[1], len(mat.data)) )
```

In [14]:

```
mat = build_matrix(docs)
mat1 = build_matrix(docs1)
csr_info(mat)
csr_info(mat1)
```

```
[nrows 60, ncols 50099, nnz 110498]
[nrows 60, ncols 47608, nnz 100737]
```

```
c:\users\ksk\opencv\lib\site-packages\ipykernel_launcher.py:20: Deprecation
Warning: `np.int` is a deprecated alias for the builtin `int`. To silence t
his warning, use `int` by itself. Doing this will not modify any behavior a
nd is safe. When replacing `np.int`, you may wish to use e.g. `np.int64` or
`np.int32` to specify the precision. If you wish to review your current us
e, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/
devdocs/release/1.20.0-notes.html#deprecations (https://numpy.org/devdocs/r
elease/1.20.0-notes.html#deprecations)
c:\users\ksk\opencv\lib\site-packages\ipykernel_launcher.py:22: Deprecation
Warning: `np.int` is a deprecated alias for the builtin `int`. To silence t
his warning, use `int` by itself. Doing this will not modify any behavior a
nd is safe. When replacing `np.int`, you may wish to use e.g. `np.int64` or
`np.int32` to specify the precision. If you wish to review your current us
e, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/
devdocs/release/1.20.0-notes.html#deprecations (https://numpy.org/devdocs/r
elease/1.20.0-notes.html#deprecations)
```

To decrease the importance of popular words in similarity computations, we usually scale the matrix by the *Inverse Document Frequency* (IDF). Furthermore, normalizing the vectors helps us compute cosine similarity more efficiently. Run the cell below to scale the `mat` matrix and create a second version with normalized row vectors. Note how the scaling and normalization are done in `O(nnz)` time.

In [36]:

```
# scale matrix and normalize its rows
def csr_idf(mat, copy=False, **kargs):
    r""" Scale a CSR matrix by idf.
    Returns scaling factors as dict. If copy is True,
    returns scaled matrix and scaling factors.
    """
    if copy is True:
        mat = mat.copy()
    nrows = mat.shape[0]
    nnz = mat.nnz
    ind, val, ptr = mat.indices, mat.data, mat.indptr
    # document frequency
    df = defaultdict(int)
    for i in ind:
        df[i] += 1
    # inverse document frequency
    for k,v in df.items():
        df[k] = np.log(nrows / float(v))  ## df turns to idf - reusing memory
    # scale by idf
    for i in range(0, nnz):
        val[i] *= df[ind[i]]

    return df if copy is False else mat

def csr_l2normalize(mat, copy=False, **kargs):
    r""" Normalize the rows of a CSR matrix by their L-2 norm.
    If copy is True, returns a copy of the normalized matrix.
    """
    if copy is True:
        mat = mat.copy()
    nrows = mat.shape[0]
    nnz = mat.nnz
    ind, val, ptr = mat.indices, mat.data, mat.indptr
    # normalize
    for i in range(nrows):
        rsum = 0.0
        for j in range(ptr[i], ptr[i+1]):
            rsum += val[j]**2
        if rsum == 0.0:
            continue  # do not normalize empty rows
        rsum = 1.0/np.sqrt(rsum)
        for j in range(ptr[i], ptr[i+1]):
            val[j] *= rsum

    if copy is True:
        return mat
mat2 = csr_idf(mat1, copy=True)
mat3 = csr_l2normalize(mat2, copy=True)
print("mat1:", mat1[15,:20].todense(), "\n")
print("mat2:", mat2[15,:20].todense(), "\n")
print("mat3:", mat3[15,:20].todense())
```

```
mat1: [[ 0.  0.  0.  0.  0.  0.  3.  0.  2.  0.  0.  1.  7.  0.  0.  0.
0.  0.
  19.  0.]]
```

```
mat2: [[0.         0.         0.         0.         0.         0.
  0.15387988 0.         0.66628889 0.         0.         0.26570317
  0.35905306 0.         0.         0.         0.         0.
  0.         0.        ]]

mat3: [[0.         0.         0.         0.         0.         0.
  0.00051804 0.         0.00224307 0.         0.         0.00089449
  0.00120876 0.         0.         0.         0.         0.
  0.         0.        ]]
```

Cosine similarity is defined as below. Using the matrices `mat1` and `mat2` , compute the cosine similarity between the 2nd and 6th rows in the respective matrices, without using a distance/similarity function from some library. You may only use scipy/numpy vector or matrix operations.

In [16]:

```
%%latex
$$cos(\mathbf{a}, \mathbf{b}) = \frac{\langle \mathbf{a},
          \mathbf{b} \rangle}{||\mathbf{a}||\ ||\mathbf{b}||}$$
```

$$cos(\mathbf{a}, \mathbf{b}) = \frac{\langle \mathbf{a}, \mathbf{b} \rangle}{||\mathbf{a}||\ ||\mathbf{b}||}$$

In [27]:

```
def cosine_sim(A, B):
    return np.dot(A,B) / (np.linalg.norm(A) * np.linalg.norm(B))
```

In [46]:

```
print("Cosine similarity between Matrix1 2nd row and 6th row : ",cosine_sim(m
```

```
Cosine similarity between Matrix1 2nd row and 6th row :  0.3322986395497961
6
```

In [47]:

```
print("Cosine similarity between Matrix1 2nd row and 6th row : ",cosine_sim(m
```

```
Cosine similarity between Matrix1 2nd row and 6th row :  0.0117497177131933
64
```

In [ ]: