Implementing non-negative Matrix Factorization in Python

Philipp Schreitmüller

November 18, 2014

Abstract

This report is part of an assignment in COMP41450 during fall term 2014. The objective of this assignment is to write a new implementation of the Euclidean distance formulation of Non-negative Matrix Factorization (NMF) as proposed by Lee & Seung[1]. The algorithm is tested with real-world data by the BBC.

Contents

1 Data source **2**

1.1 bbcnews.mtx 2

1.2 bbcnews.terms 2

2 Implementation **2**

2.1 Reading data 3

2.2 TF-IDF 4

2.3 Refining W and H 4

2.4 Computing Euclidian Distance 5

2.5 Getting the top terms 6

2.6 Putting it all together 6

3 Results & Visualisation **7**

3.1 Top terms 7

3.2 Visualisation 8

1 Data source

The assignment contains real-world data from the BBC. There are two files: *bbcnews.mtx* and *bbcnews.terms*.

1.1 bbcnews.mtx

This file describes a sparse term-document matrix in the matrix market format.

```
%%MatrixMarket matrix coordinate real general
4058 1400 161462
1 1 1.0000
1 10 1.0000
1 557 1.0000
...
```

In total the term-document matrix covers 4058 terms and 1400 articles. In total 161462 non-null entries. The indices in term column are referencing the entries of *bbcnews.items*.

1.2 bbcnews.terms

This file contains all 4058 which are referenced in the *bbcnews.mtx*. The corresponding index of each term is the line number.

```
ad
sales
boost
...
```

2 Implementation

NMF with euclidian cost function was implemented in the language *Python* using the *NumPy* package which provides efficient data structures and arrays for scientific computing. *Python* was used because it's simple, cross-platform compatible and the amount of code is small. The complete code for is contained in the file `nmf.py`.

2.1 Reading data

For reading the sample data from the file two function were written: `read_term_document()` for the term-document matrix in *bbcnews.mtx* and `read_terms()` for the terms in *bbcnews.terms*.

```
def read_term_document():
    f = open("data/bbcnews.mtx")
    counter = 0
    for line in f.readlines():
        if counter == 1:
            debug = tuple([int(elem) for elem in line.split(
                " ", 2)[:2]])
            ret = np.zeros(debug, dtype=float)
        else:
            if counter > 1:
                vals = line.replace("\n", "").split(' ', 2)
                x, y = map(int, vals[:2])
                z = float(vals[2])
                ret[x - 1, y - 1] = z
        counter += 1
    f.close()
    return ret
```

After skipping the line with the comment, the first line (`counter==1`) initialises a 2d array with zeros, the others lines are used to replace the zeros with actual values. The function for reading the terms is even easier.

```
def read_terms():
    f = open("data/bbcnews.terms")
    ret = []
    for line in f.readlines():
        ret.append(line.replace("\n", ""))
    f.close()
```

```
    return ret
```

The terms are returned in a simple list.

2.2 TF-IDF

The raw term-document matrix contains simple term frequencies. We apply *TF-IDF* on those raw values to prepare for the *NMF*-Algorithm. There are more ways to implement *TF-IDF*. In this assignment the approach of `http://www.tfidf.com/` was used:

$$TF(t,d) = \frac{f(t,d)}{\sum_{w \in d} f(w,d)}$$

$$IDF(t) = \log \frac{N}{n_t}$$

$$\text{TF-IDF}(t,d) = TF(t,d) * IDF(t)$$

where N is the number of documents and n_t is the number of document which contain the term t.

```
def tf_idf(a):
    sumTerms = [sum(colum) for colum in a.T]  # sum of all ↩
       terms in a document
    sumWord = [sum(x >= 1 for x in row) for row in a]
    numDoc = len(a)
    for i in range(len(a)):
        for j in range(len(a[i])):
            a[i, j] = (a[i, j] / sumTerms[j]) * m.log(numDoc↩
                / sumWord[i])
    return a
```

2.3 Refining W and H

Before we can refine W and H, they first must be randomly initialised. This happens in `init_wh(a, k)`. This function takes the *TF-IDF* normalized term-document matrix `a` and

the amount of clusters `k` as input. It return inital `(w,h)`.

```
def init_wh(a, k):
    a_avg = np.average(a)
    w = np.random.random(a.shape[0] * k).reshape(a.shape[0],↩
        k) * a_avg
    h = np.random.random(k * a.shape[1]).reshape(k, a.shape↩
       [1]) * a_avg
    return w, h
```

An optimisation is applied as the random values are multiplied with the average over all cells in `a`. After initialising, the matrices are iteratively refined. This happens in the following way [1]):

$$H_{a\mu} \leftarrow H_{a\mu}\frac{(W^TV)_{a\mu}}{(WW^TH)_{a\mu}} \qquad W_{ia} \leftarrow W_{ia}W_{ia}\frac{(VH^T)_{ia}}{(WHH^T)_{ia}}$$

In python it is implemented like this:

```
def iter_step(a, w, h):
    h_new = h * (np.array(np.mat(w.T) * a) / np.array(np.mat↩
       (w.T) * np.dot(w, h)))
    w_new = w * (np.dot(a, h_new.T) / np.array(np.dot(w, ↩
       h_new) * np.mat(h_new.T)))
    return w_new, h_new
```

2.4 Computing Euclidian Distance

The objective of *NMF* is the optimisation of $A \approx WH$. As a cost function the square of the *Euclidian distance* can be used.

$$\| A - B \|^2 = \sum_{ij}(A_{ij} - B_{ij})^2$$

Using *Python* the cost function can be computed in a simple way:

```
def compute_distance(a, w, h):
    temp = a - np.array(np.dot(w, h))
    temp *= temp
```

```
    return np.sum(temp)
```

2.5 Getting the top terms

After finishing *NMF* the resulting matrices `w` must be evaluated. That means that the top terms for each cluster should be named. To get the indices of the top term, the columns of matrix `w` must be sorted. That happens in `get_max_indices(w, terms)` where `terms` determines the number of top terms per cluster:

```
def get_max_indices(w, terms):
    ret = []
    for column in w.T:
        ret.append(column.argsort()[-terms:][::-1])
    return ret
```

If you map those indices on the list with the terms, you will get the corresponding terms.

2.6 Putting it all together

If you concatenate all the described steps, the desired algorithm results:

```
def nmf(a, min_delta, max_iter, k, num_terms):
    w, h = init_wh(a, k)
    best_w = w
    e = delta_e = new_e = smallest_e = compute_distance(a, w↩
       , h)
    i = 0
    while i < max_iter and delta_e > min_delta:
        w, h = iter_step(a, w, h)
        new_e = compute_distance(a, w, h)
        if new_e < smallest_e:
            smallest_e = new_e
            best_w = w
        delta_e = e - new_e
        e = new_e
```

```
        i += 1
    return get_max_indices(best_w, num_terms), i, best_w
```

Refinition of `w` and `h` stop either if the minimal difference in error falls below `min_delta` or if `max_iter` is exceeded. `nmf` can be called with different amount of clusters. The resulting `best_w` then can be examined with `get_max_indices(w, terms)`.

3 Results & Visualisation

In the following part the results of the experiments with the NMF algorithm are described.

3.1 Top terms

2 Clusters					
game	people				
england	government				
win	labour				
3 Clusters					
game	labour	people			
england	**election**	**firm**			
win	**party**	**market**			
4 Clusters					
game	labour	**oil**	**users**		
england	election	**sales**	people		
win	party	**growth**	**microsoft**		
5 Clusters					
game	labour	oil	microsoft	**mobile**	
england	election	sales	**sofware**	**phone**	
win	party	growth	users	**music**	
6 Clusters					
game	labour	sales	microsoft	mobile	**yukos**
england	election	growth	sofware	phone	oil
win	party	market	users	music	**russian**

In the table above the results of the clustering are shown (2000 Iterations). In each cluster the top **3** terms are listed:

New terms are written in **bold**. Especially interesting is the fact that new clusters consist of almost only new terms e.g. {oil, sales, growth} or {mobile, phone, music}. The algorithm did a good job as the terms fit to their clusters quite good.

3.2 Visualisation

Some visualisation was done on the cost function: Obviously the error decreases the more

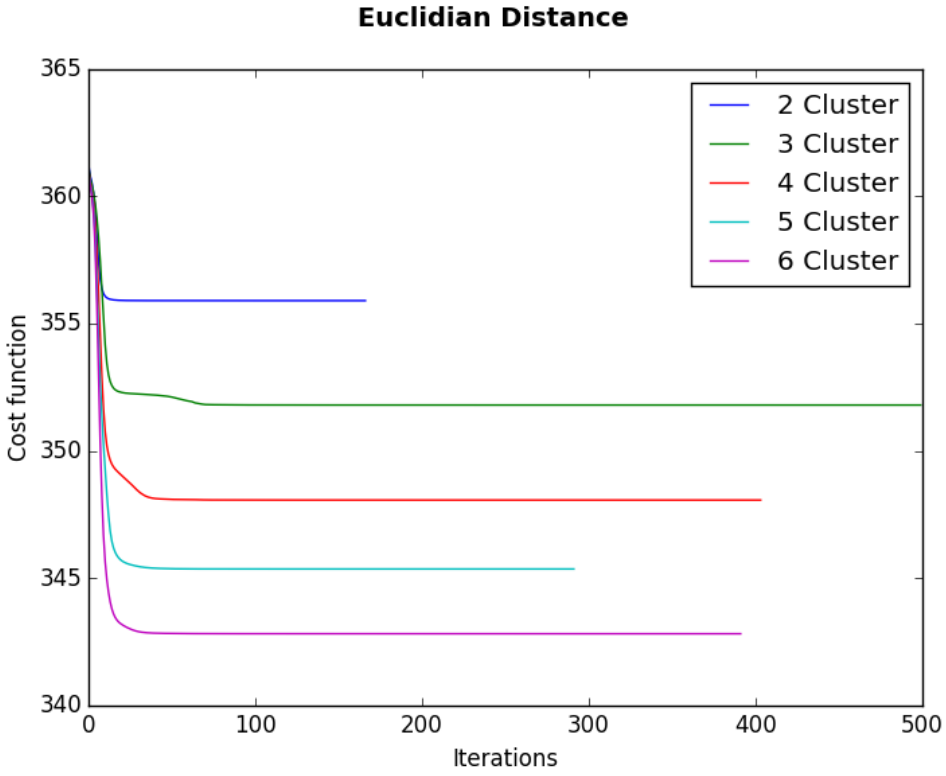

Figure 1: Development of square Euclidian distance.

clusters are computed.

As part of the experiments `w` and `h` where initialised with ones. The results are much worse than for random initialised matrices. There are also duplicated clusters in the results.

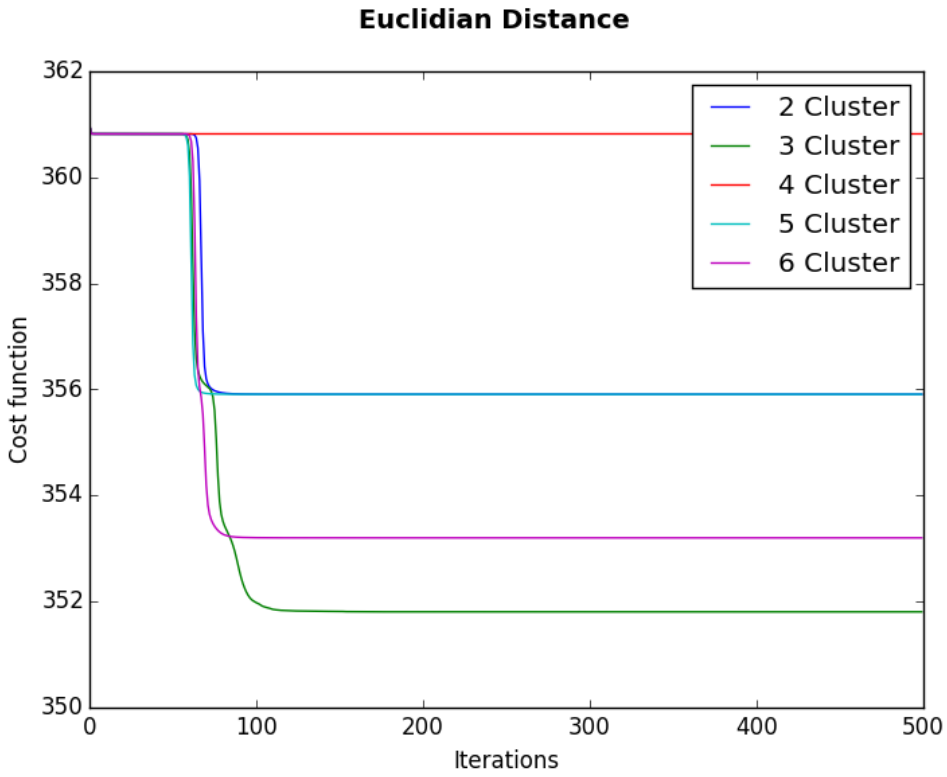

Figure 2: Development of square Euclidian distance with w and h initialised with ones .

Visualisation was also done on the basis matrix for 2 and for 6 Clusters. The scale for 6 clusters goes up to 0.030 where it's only 0.012 for 2 clusters.

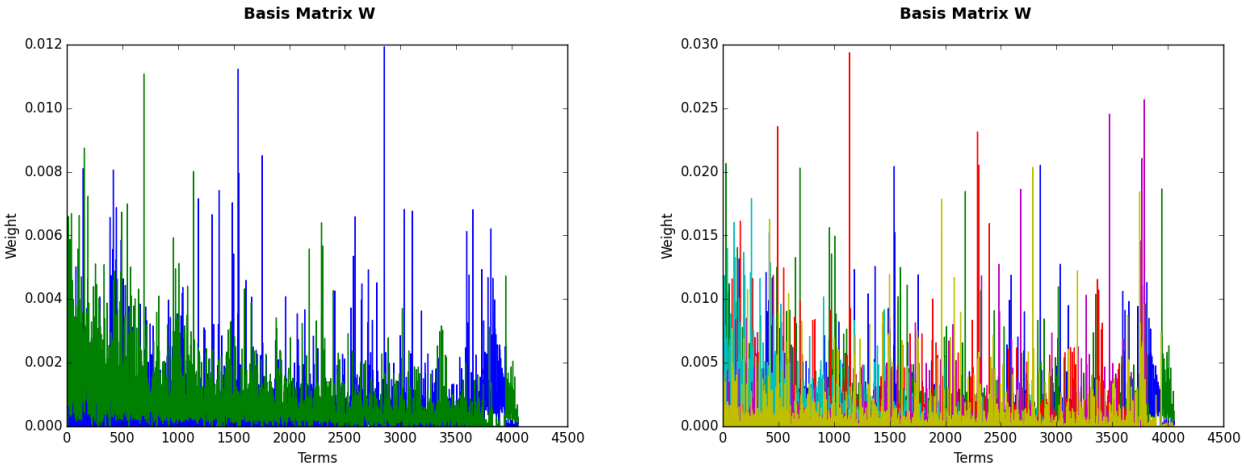

Figure 3: Basis matrix for 2 and 6 clusters.

References

[1] Daniel D. Lee and H. Sebastian Seung. Algorithms for non-negative matrix factorization. In T.K. Leen, T.G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 556–562. MIT Press, 2001.