lab4-NMFfrancoisamat

November 29, 2018

In [1]:
```
# -*- coding: utf-8 -*-
  # Author: Olivier Grisel <olivier.grisel@ensta.org>
# Lars Buitinck
# Chyi-Kwei Yau <chyikwei.yau@gmail.com> # License: BSD 3 clause

from __future__ import print_function
from time import time
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.decomposition import NMF
from sklearn.datasets import fetch_20newsgroups
import numpy as np
import math
import sys
```

In [2]:
```
DEBUG = False
Verbose = DEBUG

n_samples = 2000
n_features = 1000
n_components = 10
n_top_words = 20
```

In [3]:
```
def print_top_words(model, feature_names, n_top_words,Verbose = True):
    array_of_message = []
    for topic_idx, topic in enumerate(model.components_):
        message = "Topic #%d: " % topic_idx
        message += " ".join([feature_names[i] for i in topic.argsort()[:-n_top_words -
        array_of_message.append(message)
        if(Verbose):
            print(message)
    if(Verbose):
        print()
    return array_of_message
```

In [4]:
```
def load_data_newspaper(Verbose = False):
    # Load the 20 newsgroups dataset and vectorize it. We use a few
    # heuristics to filter out useless terms early on: the posts are
    # stripped of headers, footers and quoted replies, and common
```

```
            # English words, words occurring in only one document or in at
            # least 95% of the documents are removed.
            if(Verbose== True):
                print("Loading dataset...")
            t0 = time()
            dataset = fetch_20newsgroups(shuffle=True, random_state=1,
                                         remove=('headers', 'footers', 'quotes'))
            data_samples = dataset.data[:n_samples]
            if(Verbose):
                print("done in %0.3fs." % (time() - t0))
            return dataset, data_samples
```

```
In [5]: def create_tfidf(Verbose=False):
            # Use tf-idf features for NMF.
            if(Verbose):
                print("Extracting tf-idf features...")

            tfidf_vectorizer = TfidfVectorizer(max_df=0.95, min_df=2,
                                               max_features=n_features,
                                               stop_words='english')
            t0 = time()
            tfidf = tfidf_vectorizer.fit_transform(data_samples)
            if(Verbose== True):
                print("done in %0.3fs." % (time() - t0))

            return tfidf_vectorizer, tfidf
```

```
In [6]: def print_messages(i):
            print(random_messages[i]) # sqrt(X.mean() / n_components)
            print(nndsvd_messages[i])  # better sparseness
            print(nndsvda_messages[i]) # when sparsity is not desired
            print(nndsvdar_messages[i]) # faster, less accurate alternative to NNDSVDa
        def print_messages_kull(i):
            print(random_messages_kull[i]) # sqrt(X.mean() / n_components)
            print(nndsvd_messages_kull[i])  # better sparseness
            print(nndsvda_messages_kull[i]) # when sparsity is not desired
            print(nndsvdar_messages_kull[i]) # faster, less accurate alternative to NNDSVDa
        def print_messages_it(i):
            print(random_messages_it[i]) # sqrt(X.mean() / n_components)
            print(nndsvd_messages_it[i])  # better sparseness
            print(nndsvda_messages_it[i]) # when sparsity is not desired
            print(nndsvdar_messages_it[i]) # faster, less accurate alternative to NNDSVDa
```

```
In [7]: def check_diff(a,b,verbose_name):
            for i in range(len(a)):
                if (a[i] != b[i]):
                    print("they are different" + " "+ verbose_name)
                    return False
```

```
In [8]: def printErrorAndIter(dictOfNmf):
            print("iterations \t","error\t\t","name")
            for nmf in dictOfNmf.keys():
                print(nmf.n_iter_,"\t \t",nmf.reconstruction_err_, " nmf :", dictOfNmf[nmf])

In [9]: dataset, data_samples = load_data_newspaper()
        tfidf_vectorizer, tfidf = create_tfidf()

In [10]: def build_NMF(init = "random",fixed_W = None, random_state = 1, beta_loss='frobenius'
                       tfidf=tfidf, tfidf_vectorizer=tfidf_vectorizer, Verbose=False):
             # Fit the NMF model
             if(Verbose):
                 print("Fitting the NMF model (Frobenius norm) with tf-idf features, " "n_sampl
             t0 = time()
             if(init != "custom"):
                 nmf = NMF(n_components=n_components,init = init,solver=solver, random_state =
             else:
                 fixed_H = fixed_W.T
                 if(Verbose):
                     print(fixed_W.shape)
                     print(fixed_H.shape)
                 #H : array-like, shape (n_components, n_features)
                 nmf = NMF(n_components=n_components,init = init,solver=solver, random_state =

             #init : random | nndsvd | nndsvda | nndsvdar | custom
             # random_state is the seed used by the random number generator
             # alpha is Constant that multiplies the regularization terms
             # l1_ratio The regularization mixing parameter between the l1 and l2 norm
             if(Verbose):

                 print("done in %0.3fs." % (time() - t0))
                 print("\nTopics in NMF model (Frobenius norm):")

             tfidf_feature_names = tfidf_vectorizer.get_feature_names()
             array_of_message = print_top_words(nmf, tfidf_feature_names, n_top_words,Verbose)

             return nmf, array_of_message

In [11]: nmf_random, random_messages = build_NMF(Verbose=Verbose)
         nmf_nndsvd, nndsvd_messages = build_NMF(init ="nndsvd", Verbose=Verbose)
         nmf_nndsvda, nndsvda_messages = build_NMF(init ="nndsvda", Verbose=Verbose)
         nmf_nndsvdar, nndsvdar_messages = build_NMF(init ="nndsvdar", Verbose=Verbose)

In [12]: W = np.random.rand(n_samples,n_components)
         # print(np.all(np.isfinite(W)))
         # W : array-like, shape (n_samples, n_components)
         # Custom_messages = build_NMF(init ="custom",fixed_W = W, Verbose=False)[1]
         # there is a problem with the shape of W, H that i did not understand :
         # I used the shapes told in the documentation of sklearn
```

0.0.1 1 Test and comment on the effect of varying the initialisation, especially using random nonnegative values as initial guesses (for W and H coefficients, using the notations introduced during the lecture).

```
In [13]: dictOfNmf = {nmf_random:"nmf_random", nmf_nndsvdar:"nmf_nndsvdar", nmf_nndsvda:"nmf_n
         printErrorAndIter(dictOfNmf)
```

```
iterations          error                  name
104                   42.183446447525256  nmf : nmf_random
110                   42.13858929293552  nmf : nmf_nndsvdar
106                   42.13858585218299  nmf : nmf_nndsvda
128                   42.1386080858152  nmf : nmf_nndsvd
```

Using the differents options of the init from the sklearn library, I obtain that only random gives a very different results from the others ones that have very close results for all the topics.

I notice that the error is higher for the random init and all the others methods have the same error (with 10^-4 in accuracy). Besides, we notice that the nmf_nndsvdar is faster in term of number of iteration than the nmf_nndsvd, as explained in the documentation.

In addition, we get results that vary a lot between the random and the others, especially in the the Topics {1,3,4,9}.

0.1 2. Compare and comment on the difference between the results obtained with l2 cost com- pared to the generalised Kullback-Liebler cost.

```
In [14]: nmf_random_kull,random_messages_kull = build_NMF( beta_loss="kullback-leibler",solver=
         nmf_nndsvd_kull,nndsvd_messages_kull = build_NMF(init ="nndsvd",beta_loss="kullback-l
         nmf_nndsvda_kull,nndsvda_messages_kull = build_NMF(init ="nndsvda",beta_loss="kullbac
         nmf_nndsvdar_kull,nndsvdar_messages_kull = build_NMF(init ="nndsvdar",beta_loss="kull
```

```
/usr/local/lib/python3.6/site-packages/sklearn/decomposition/nmf.py:212: UserWarning: The multi
  UserWarning)
```

```
In [15]: dictOfNmf_kull = {nmf_random_kull :"nmf_random_kull", nmf_nndsvdar_kull:"nmf_nndsvdar_
         printErrorAndIter(dictOfNmf_kull)
```

```
iterations          error                  name
170                   212.11368422840948  nmf : nmf_random_kull
130                   211.17330591792012  nmf : nmf_nndsvdar_kull
100                   211.11744121234366  nmf : nmf_nndsvda_kull
60                   214.08809055091118  nmf : nmf_nndsvd_kull
```

```
In [16]: nmf_random_it, random_messages_it = build_NMF(beta_loss="itakura-saito",solver='mu',Ve
         nmf_nndsvd_it, nndsvd_messages_it = build_NMF(init ="nndsvd",beta_loss="itakura-saito"
         nmf_nndsvda_it, nndsvda_messages_it = build_NMF(init ="nndsvda",beta_loss="itakura-sa
         nmf_nndsvdar_it, nndsvdar_messages_it = build_NMF(init ="nndsvdar",beta_loss="itakura-
```

```
/usr/local/lib/python3.6/site-packages/sklearn/decomposition/nmf.py:156: RuntimeWarning: invali
  return np.sqrt(2 * res)
/usr/local/lib/python3.6/site-packages/sklearn/decomposition/nmf.py:1035: ConvergenceWarning: I
  " improve convergence." % max_iter, ConvergenceWarning)
/usr/local/lib/python3.6/site-packages/sklearn/decomposition/nmf.py:212: UserWarning: The multi
  UserWarning)
```

```
In [17]: dictOfNmf_it = {nmf_random_it :"nmf_random_it", nmf_nndsvdar_it:"nmf_nndsvdar_it", nmf
         printErrorAndIter(dictOfNmf_it)
```

```
iterations          error                  name
200                   nan  nmf : nmf_random_it
200                   nan  nmf : nmf_nndsvdar_it
200                   nan  nmf : nmf_nndsvda_it
200                   nan  nmf : nmf_nndsvd_it
```

The first results, obtained in the first question are computed with the l2 cost. As observed here, we have around 5 times more error with every methods using Kullback-Liebler than using the l2 cost. (42 vs 211).

In addition, all algorithms take fewer steps with the l2 norm execpt for the nndsvd. for the nndsvd, the number of operation has been divided by two.

Observing the themes found by these methods, the themes are similar but the Kullback-Liebler seems less accurate.

Finally, When using the itakura-saito method, I observe that the algorithm reach its max_iter limit at 200, as seen in the course, we cannot predict if this method converge or not, it doesn't here.

0.1.1 3. Test and comment on the results obtained using a simpler term-frequency representation as input (as opposed to the TF-IDF representation considered in the code above) when considering the Kullback-Liebler cost.

```
In [18]: nmf_random_kull, random_messages_kull = build_NMF( beta_loss="kullback-leibler",solver
         #tfidf_vectorizer = TfidfVectorizer(max_df=0.95, min_df=2,max_features=n_features,stop
         vectorizer = CountVectorizer(max_df = 0.95, min_df = 2, max_features = n_features, sto
         features = vectorizer.fit_transform(data_samples)
         nmf_random_kull_count, random_messages_kull_count = build_NMF(tfidf=features, beta_los
```

```
In [19]: dictOfNmf_kull_test = {nmf_random_kull:"kull with tdidf",nmf_random_kull_count:"kull w
         printErrorAndIter(dictOfNmf_kull_test)
```

```
iterations          error                  name
170                   212.11368422840948  nmf : kull with tdidf
190                   592.5776601956052  nmf : kull with countVectorizer
```

The simplier term frequency representation as countvectorizer, with the Kullback-Liebler cost, get a much highier error, with more iterations: around 3 times for the error and 20 iterations more (212 vs 592 and 170 vs 190).

```
In [20]: ### TEST code for the first part
         if DEBUG :
             for i in range(len(random_messages_it)):
                 print_messages(i)
                 print()
                 print_messages_it(i)
                 print()
             for i in range(len(nndsvdar_messages_kull)):
                 print("TOPIC %d" %i)
                 print_messages_kull(i)
                 print()
                 print()
                 print_messages(i)
                 print()
                 print()
             for i in range(len(nndsvdar_messages)):
                 print("TOPIC %d" %i)
                 print_messages(i)
                 print()
                 print()
             print(check_diff(random_messages_kull,random_messages,"random_messages kull"),
                 check_diff(nndsvd_messages_kull,nndsvd_messages,"nndsvd_messages kull "),
                 check_diff(nndsvda_messages_kull,nndsvda_messages,"nndsvda_messages kull"),
                 check_diff(nndsvdar_messages_kull,nndsvdar_messages,"nndsvdar_messages kull"))

             print(check_diff(random_messages_it, random_messages, "random_messages it"),
             check_diff(nndsvd_messages_it, nndsvd_messages, "nndsvd_messages it "),
             check_diff(nndsvda_messages_it, nndsvda_messages, "nndsvda_messages it"),
             check_diff(nndsvdar_messages_it, nndsvdar_messages, "nndsvdar_messages it"))
```

0.1.2 - CUSTOM NMF IMPLEMENTATION -

Implement the multiplicative update rules (derived from the majorisation-minimisation approach) for NMF estimation with divergences, including the case = 1 (generalised Kullback-Liebler divergence). Ensure that :

1. you can easily choose a custom initialisation for the W and H matrices ;
2. you can set a custom number of iteration ;
3. you can monitor the behaviour of the loss function across the iterations and that it is readily decreasing. Compare your implementation with the one offered by scikit-learn.

```
In [21]: class Custom_nmf :
```

```
def __init__(self, features, beta,W,H, k=2,tole=0.01):
    self.k = k
    self.tole = tole
    self.beta = beta

    self.features = features

    self.W = np.random.rand(features.shape[0],k)
    self.H = np.random.rand(k,features.shape[1])

    if(beta == 0 ):
        self.betafunction = self.itakura_saito
    elif(beta == 1):
        self.betafunction = self.kullback_leiber
    else :
        self.betafunction = self.euclidean_distance


def euclidean_distance(self, x, y, beta):
    return (1 / ( beta*( beta - 1) ) )*(np.pow(x, beta) + (beta - 1)*np.pow(y, bet

def kullback_leiber(self, x, y, beta):
    return x * np.log(x / y) - x + y

def itakura_saito(self, x, y, beta):
    return (x / y) - np.log( x / y) - 1

def get_error(self):
    W,H,features = self.W,self.H,self.features
    function =  self.betafunction
    WH = np.dot(W, H)
    err = 0
    for i in range(features.shape[0]):
        for j in range(features.shape[1]):
            x = features[:,j][i][0]
            x = np.squeeze(np.asarray(x))
            y = WH[i][j]
            if (x == 0 or np.isnan(x)) :
                break
            if(y == 0 or np.isnan(y)):
                break
            #x = sys.float_info.epsilon
            #y = np.squeeze(np.asarray(WH[i][j]))
            #y = WH[:,j][i][0]
            #print("y:",y,"WH:", WH)
            #if(y == 0 ):
            #    break
            res = function(x,y,beta)
```

```
                    if( not np.isnan(res)):
                        err += function(x,y,beta)
        return err


    def nmf(self,max_iter=200):
        W,H,features,beta = self.W,self.H,self.features,self.beta
        init_err = self.get_error()
        err = init_err
        for it in range(max_iter):
            W,H = self.W, self.H
            WH = np.dot(W,H)
            WH_BETA  = np.power(WH,beta-2)
            num = np.dot(W.T, np.multiply(WH_BETA, features))
            dem = np.dot(W.T, np.power(WH,beta-1))
            term = np.divide(num, dem)
            H = np.multiply(H, term)

            self.H = np.squeeze(np.asarray(H))
            print(self.H)

            W,H = self.W, self.H
            WH = np.dot(W,H)
            WH_BETA  = np.power(WH,beta - 2)
            num = np.dot(np.multiply(WH_BETA, features),H.T)
            dem = np.dot(np.power(WH,beta - 1), H.T)
            term = np.divide(num, dem)
            W = np.multiply(W, term)
            self.W = np.squeeze(np.asarray(W))



            current_err = self.get_error()
            progres = ((err - current_err) / init_err)
            print("err: ",current_err,"it:",it, "err_init: ", init_err)
            print("progres: ",progres,"it:",it)

            err = current_err
            if(progres < self.tole and progres > 0):
                break

        return self.W, self.H
```

In [22]:
```
features = tfidf.todense()

beta = 1
W, H = None, None
customNMF = Custom_nmf(features, beta, W, H)
```

```
        print(customNMF.get_error())
```

```
16.87037313844045
```

```
In [23]: customNMF.nmf(max_iter =3)
```

```
[[0.00702013 0.0050419  0.00831683 ... 0.00630791 0.00952164 0.00273914]
 [0.00595883 0.00350593 0.00866043 ... 0.01338394 0.00557994 0.00170405]]
err:  44.89977853682493 it: 0 err_init:  16.87037313844045
progres:  -1.6614573470528233 it: 0
[[nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]]
err:  0 it: 1 err_init:  16.87037313844045
progres:  2.6614573470528233 it: 1
```

```
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:64: RuntimeWarning: divide by zer
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:65: RuntimeWarning: invalid value
```

```
[[nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]]
err:  0 it: 2 err_init:  16.87037313844045
progres:  0.0 it: 2
```

```
Out[23]: (array([[nan, nan],
                 [nan, nan],
                 [nan, nan],
                 ...,
                 [nan, nan],
                 [nan, nan],
                 [nan, nan]]), array([[nan, nan, nan, ..., nan, nan, nan],
                 [nan, nan, nan, ..., nan, nan, nan]]))
```

I created a programm that conforms with the first two obligations (set up W, H and set a custom number of iteration). It's easely setable with this implementation with object programming.

But I still have problems with the actualisation of the W,H matrix, the convertion of csr_matrix to ndarray seems to convert W,H to nadrray of nan that make the algortithm useless after 2 iterations. However, in this case, after two iterations the error decrease.