GSoC '17 Project Proposal
Online Non Negative Factorisation

Krishna Kant Singh

March 2017

1 Basic Information

Name and Contact Info

KrishnaKant Singh(Kris)

Email:cs15mtech11007@iith.ac.in

IRC Nickname: kris1

Phone +918895952597

Github/Gitter: kris-singh

Video-chat: www.appear.in/kris-singh

University and Current Enrollment

University : Indian Institute of Technology, Hyderabad

Field of Study: Computer Science(2018)

Coding Skills

Python : Fluent, Good knowledge of OOPS

C++: Fluent, Good knowledge of Meta Programing, OOPS, Policy based design through templates

Cython: Beginner but can come up to speed in quick time.

Development Environment

Ubuntu/Linux variants

Ides: Eclipse, PyCharm, Sublime

Good Familiarity with conda/venv

Version Control: Fluent with Git

2 Problem Statement/ Background

Given a matrix we wish to factorise the matrix $X \in R^{r \times d}$ into the product of 2 full rank matrices $\mathcal{F} \in \mathcal{R}^{d \times r}$ and $\mathcal{G} \in \mathcal{R}^{n \times r}$ subject to

$$\mathcal{L}(\mathcal{F}, \mathcal{G}) = (||\mathcal{X} - \mathcal{F}\mathcal{G}^{\mathcal{T}}||_{\mathcal{F}}{}^{2}) \tag{1}$$

$|| \; ||_F$ indicates the Frobenius norm.
NMF finds usage in the document clustering and recommendation system among many other areas. Many algorithms like word2vec and Probabilistic LSI can also be derived/ thought of as a form of NMF. But the most useful/immediate aspect for gensim is in the text mining domain wherein it is used as a tool to cluster the document into different categories.
There exist a couple of standard ways of solving the given problem. Projected Gradient Descent Model and HALS(Hierarchical Alternating Least Square). All these methods can shown to be derived from the Block Coordinate Descent Method. We give a brief overview of these algorithms.

General NMF Algorithm

Initialise the matrices F and G;
while *Converge* **do**
  1.Optimise the function $min_{F \in \mathcal{R}_{+}^{d \times r}, F \in \mathcal{R}_{+}^{n \times r}} \mathcal{L}(F, G)$;
  2.Update F and G ;
end

Algorithm 1: General NMF Algorithm

For Initialization The most popularly used method is Non Negative Singular Value Decomposition and its variants.

NNSVD[1] uses a very simple approach that utilises the idea of SVD. Since SVD gives the best k-rank approximation to any matrix on forbenius norm we utilise this fact and use sum of rank on matrices to initialize the the F and G matrix. Other variants are NNSVDa and NNSVDvar which fill the zeros values in F and G by the average of all the values and random values respectively.

2.1 Projected Gradient Descent

Tries to convert the non-convex objective to a convex objective using alternating minimization. This is achieved by minimising one parameter at a time keeping the other constant.

$$F_k + 1 = P[F_k - \alpha_k \nabla L(F_k, G^{(t)})] \tag{2}$$

The gradient is computed with respect to F at F_k keeping G constant. $G^{(t)}$ indicates the value of G at the t iteration. We do the same thing for updating the G matrix but keeping F as a constant.

2.2 Hierarchical Alternating Least Square

Projected Gradient Descent is not guaranteed to converge and also if it converges it does so very slowly $O(Tndr + TKJr_2(d+n))$. Where T is the number of iterations, K is the average number of PGD iterations for updating F or G in one round, and J is the average number of trials needed for implementing the Armijo rule for updating the step size.

Hierarchical ALS uses the ALS manner with minor trick we first define the ALS method

2.3 Alternating Non-Negative Least Square

1. Initialise matrix using NNSVD
2. Estimate G from the matrix equation $F^TFG = F^TX$ by solving

$$min_G D_F(X||FG) = \frac{1}{2}||X - FG||_F^2 \tag{3}$$

   with F fixed
3. Set all negative values to 0
4. Estimate F from the matrix equation $GG^TF^T = GX^T$ by solving

$$min_F D_F(X||FG) = \frac{1}{2}||X - FG||_F^2 \tag{4}$$

   with G fixed
5. Set all negative values to 0
   Thus we can say G $\longleftarrow \max\{\epsilon, (F^TF)^{-1}F^TX\}$
   F $\longleftarrow \max\{\epsilon, XG^T(GG^T)^{-1}\}$

2.4 HALS

Hierarchical ALS makes use of simple insight. When the Dimension of the input matrix X is very high we can safely assume that it low rank. And we do not need to process all the entries of the matrix in order to estimate the factor the matrix into F and G. We can consider alternating factorization of much smaller dimension

$$X_r = F_rG + E_r \tag{5}$$

For a fixed(known) F_r

$$X_c = FG_c + E_c \tag{6}$$

For a fixed(known) G_c where $X_r \in R_+^{R\times k}$ and $X_c \in R_+^l \times X$ are matricies constructed from preselected rows and columns of the data matrix. Similarly

we can construct a low dimensional F and G factor matricies. We can plugin the valuess for the reduced dimension F and G to get the follow update rules

$$F \longleftarrow [X_c G_c^T (G_c G_c)^- 1]_+ \tag{7}$$

$$G \longleftarrow [(F_r^T F_r)^- 1 A_r^T X_r]_+ \tag{8}$$

We can further optimise the implementation using residues for ANLS rather than the actual Data matrix. For indepth explanation please refer [2] We now give the algorithms for both the HALS and FAST HALS.

Algorithm 2 FAST HALS for NMF: $\boldsymbol{Y} \approx \boldsymbol{A}\boldsymbol{B}^T$

1: Initialize nonnegative matrix $\boldsymbol{A}$ and/or $\boldsymbol{B}$ using ALS
2: Normalize the vectors $\boldsymbol{a}_j$ (or $\boldsymbol{b}_j$) to unit ℓ_2-norm length
3: **repeat**
4: % Update $\boldsymbol{B}$;
5: $\boldsymbol{W} = \boldsymbol{Y}^T \boldsymbol{A}$;
6: $\boldsymbol{V} = \boldsymbol{A}^T \boldsymbol{A}$;
7: **for** $j = 1$ to J **do**
8: $\boldsymbol{b}_j \Leftarrow \left[\boldsymbol{b}_j + \boldsymbol{w}_j - \boldsymbol{B}\, \boldsymbol{v}_j\right]_+$
9: **end for**
10: % Update $\boldsymbol{A}$;
11: $\boldsymbol{P} = \boldsymbol{Y}\boldsymbol{B}$;
12: $\boldsymbol{Q} = \boldsymbol{B}^T \boldsymbol{B}$;
13: **for** $j = 1$ to J **do**
14: $\boldsymbol{a}_j \Leftarrow \left[\boldsymbol{a}_j\, q_{jj} + \boldsymbol{p}_j - \boldsymbol{A}\, \boldsymbol{q}_j\right]_+$
15: $\boldsymbol{a}_j \Leftarrow \boldsymbol{a}_j / \|\boldsymbol{a}_j\|_2$;
16: **end for**
17: **until** convergence criterion is reached

Algorithm 1 HALS for NMF: Given $\boldsymbol{Y} \in \mathbb{R}_+^{I \times K}$ estimate $\boldsymbol{A} \in \mathbb{R}_+^{I \times J}$ and $\boldsymbol{X} = \boldsymbol{B}^T \in \mathbb{R}_+^{J \times K}$

1: Initialize nonnegative matrix $\boldsymbol{A}$ and/or $\boldsymbol{X} = \boldsymbol{B}^T$ using ALS
2: Normalize the vectors $\boldsymbol{a}_j$ (or $\boldsymbol{b}_j$) to unit ℓ_2-norm length,
3: $\boldsymbol{E} = \boldsymbol{Y} - \boldsymbol{A}\boldsymbol{B}^T$;
4: **repeat**
5: **for** $j = 1$ to J **do**
6: $\boldsymbol{Y}^{(j)} \Leftarrow \boldsymbol{E} + \boldsymbol{a}_j \boldsymbol{b}_j^T$;
7: $\boldsymbol{b}_j \Leftarrow \left[\boldsymbol{Y}^{(j)T}\, \boldsymbol{a}_j\right]_+$
8: $\boldsymbol{a}_j \Leftarrow \left[\boldsymbol{Y}^{(j)}\, \boldsymbol{b}_j\right]_+$
9: $\boldsymbol{a}_j \Leftarrow \boldsymbol{a}_j / \|\boldsymbol{a}_j\|_2$;
10: $\boldsymbol{E} \Leftarrow \boldsymbol{Y}^{(j)} - \boldsymbol{a}_j \boldsymbol{b}_j^T$;
11: **end for**
12: **until** convergence criterion is reached

Figure 1: HALS and FastHals implementation. These both algorithms can also be thought of as multiplicative updates using Euclidean Distance.

3 Existing Solutions

3.1 Scikit Learn

Scikit Learn at present has support for both Projected Gradient Descent[3] and the HALS [4] Algorithm. Though both the algorithms are neither online nor distributive.

For initialisation of the Factor matrices, Scikit learn uses the aforementioned NNSVD, NNSVDA, NNSVDVar. Scikit doesn't provide a way for choosing the number of components in the factors.

3.2 LibMF

Libmf[5]has implementations for the Hogwild Algorithm[5] and the DSGD algorithm in c++. We explain about both the algorithms below.

$$min_{p,q} \sum_{u,v} ((r_u, v - p_u^T q_v)^2 + \lambda_p ||p_u||^2 + \lambda_q ||q_v||^2) \tag{9}$$

r_u, v indicates the u,v th entry in the Input matrix R. p_u and q_v are the uth and vth column of factor matrices u and v. The other terms act as regularisers for the values in the factor matrices controlled by the α parameter. Using Gradient Descent Optimisation Technique we can update the factor matrices as follows.

$$p_u \longleftarrow p_u + \gamma(e_{u,v}q_v - \lambda_P p_u) \tag{2}$$

$$q_v \longleftarrow q_v + \gamma(e_{u,v}p_u - \lambda_Q q_v) \tag{3}$$

where $e_{u,v} = r_{u,v} - p_u^T q_v$ The HogWild algorithm is both out of core and both

Algorithm 1 HogWild's Algorithm

Require: number of threads s, $R \in \mathbb{R}^{m \times n}$, $P \in \mathbb{R}^{k \times m}$, and $Q \in \mathbb{R}^{k \times n}$
1: **for each** thread i parallelly **do**
2: **while** true **do**
3: randomly select an instance $r_{u,v}$ from R
4: update corresponding $\boldsymbol{p}_u$ and $\boldsymbol{q}_v$ using (2)-(3), respectively
5: **end while**
6: **end for**

Figure 2: HogWild Algorithm

online. But the major problem with HogWild algorithm is that it can lead to double updates of the same $r_{u,v}$ value. Because this reason convergence cannot be guaranteed in every setting.

3.2.1 DSGD

Algorithm 2 $[\mathbf{W}, \mathbf{H}]$ = Naive-Parallel-NMF $(\mathbf{A}, k)$

Require: $\mathbf{A}$ is an $m \times n$ matrix distributed both row-wise and column-wise across p processors, k is rank of approximation
Require: Local matrices: $\mathbf{A}_i$ is $m/p \times n$, $\mathbf{A}^i$ is $m \times n/p$, $\mathbf{W}_i$ is $m/p \times k$, $\mathbf{H}^i$ is $k \times n/p$
1: p_i initializes $\mathbf{H}^i$
2: **while** stopping criteria not satisfied **do**
/* **Compute W given H** */
3: collect $\mathbf{H}$ on each processor using all-gather
4: p_i computes $\mathbf{W}_i \leftarrow \underset{\tilde{\mathbf{W}} \geqslant 0}{\operatorname{argmin}} \|\mathbf{A}_i - \tilde{\mathbf{W}}\mathbf{H}\|$
/* **Compute H given W** */
5: collect $\mathbf{W}$ on each processor using all-gather
6: p_i computes $\mathbf{H}^i \leftarrow \underset{\tilde{\mathbf{H}} \geqslant 0}{\operatorname{argmin}} \|\mathbf{A}^i - \mathbf{W}\tilde{\mathbf{H}}\|$
7: **end while**
Ensure: $\mathbf{W}, \mathbf{H} \approx \underset{\tilde{\mathbf{W}} \geqslant 0, \tilde{\mathbf{H}} \geqslant 0}{\operatorname{argmin}} \|\mathbf{A} - \tilde{\mathbf{W}}\tilde{\mathbf{H}}\|$
Ensure: $\mathbf{W}$ is an $m \times k$ matrix distributed row-wise across processors, $\mathbf{H}$ is a $k \times n$ matrix distributed column-wise across processors

Figure 3: Naive Parallel NMF

1 The DSGD[6] algorithm naively divides the input matrix into s*s blocks

Algorithm 2 DSGD's Algorithm

Require: number of threads s, maximum iterations T, $R \in \mathbb{R}^{m \times n}$, $P \in \mathbb{R}^{k \times m}$, and $Q \in \mathbb{R}^{k \times n}$

```
grid R into s × s blocks B and generate s patterns covering all blocks
for t = {1, ..., T} do
    Decide the order of s patterns sequentially or by random permutation
    for each pattern of s independent blocks of B do
        assign s selected blocks to s threads
        for b = {1, ..., s} parallellly do
            randomly sample ratings from block b
            apply (2)-(3) on all sampled ratings
        end for
    end for
end for
```

Figure 4: HogWild Algorithm

and then assigns these to s threads. The updates generally are performed using the gradient descent approach same as in HogWild Approach. The method deals with the problem of overwrites of the hog wild approach but at the expense of communication between the threads.

3.3 Nmf libraray

This includes c++ parallel implementation of the HALS, Multiplicative Updates, Block Pivot Partition algorithm. Pros: Is faster that MU. Has good results based on speed Cons: Harder to implement.

3.4 Beta Divergence Multiplicative Update

The beta Divergence Multiplicative update library[7] provides both an online manner(batch nmf) and out of core computation(theano based). The paper describes multiplicative updates that can be done in an online fashion for various kind of update schemes. Also it multiplicative updates to a general loss function the Beta divergence. Where $\beta = 2$ is equivalent to euclidean loss $\beta = 1$ is equivalent to generalised kl divergence.

Vanilla Multiplicative Update

$$min_{W,H} D(V|WH) \tag{10}$$

Where is the given matrix and W,H are the factor matrices.
D is beta divergence loss.
The multiplicative updates for solving above problem are defined below

$$H \longleftarrow H \odot \frac{W^T[(WH)^{\beta-2} \odot V]}{W^T(WH)^{\beta-1}} \tag{11}$$

$$W \longleftarrow W \odot \frac{[(WH)^{\beta-2} \odot V]H^T}{(WH)^{\beta-1}H^T} \tag{12}$$

As we can clearly see from the above that the updating of H requires a full V

Algorithm 3 Basic alternating scheme for MU rules

Require: $\mathbf{V} \in \mathbb{R}_+^{F \times N}$, β, max_iter
1: Initialise **H**, **W** with nonnegative random coefficients
2: **for** $it = 0$; $it <$ max_iter **do**
3: Update **H** with (10)
4: Update **W** with (11)
5: **end for**
6: **return H**, **W**

Figure 5: Naive MU updates

which is not feasible in many cases eg online learning system. The key insight is the separability of the cost function

$$D(V|WH) = \sum_{f=1}^{F} \sum_{n=1}^{N} d(V_{f,n} W H_{f,n}$$

where d represents the scalar cost function. Hence we can update the H factor based on only a single batch of V rather that requiring the full V matrix. Now all that remains is how do we update the W factor. There are several algorithms proposed for the updating of W we show here a few. **Pros**: Easy to implement, Online and out of core
Cons: Relies on external Library like Theano for out of core compatibility, Multiplicative Updates have shown to have a poor performance then FastHals[2] and other such methods when they are implemented out of core.

Algorithm 4 Cyclic mini-batch for MU rules

Require: $\mathbf{V} \in \mathbb{R}_+^{F \times N}$, β, max_epoch
1: Initialise **H**, **W** with nonnegative random coefficients
2: **for** $ep = 0$; $ep <$ max_epoch **do**
3: Initialise $\nabla^- \mathbf{W} = 0$ and $\nabla^+ \mathbf{W} = 0$
4: **for** $b = 1$; $b < B$ **do**
5: Update $\mathbf{H}_b$ with (10)
6: $\nabla^- \mathbf{W}$ += $\nabla^- \mathbf{W}_b$
7: $\nabla^+ \mathbf{W}$ += $\nabla^+ \mathbf{W}_b$
8: **end for**
9: $\mathbf{W} \leftarrow \frac{\nabla^- \mathbf{W}}{\nabla^+ \mathbf{W}}$
10: **end for**
11: **return H**, **W**

Algorithm 5 Asymmetric SG mini-batch MU rules (ASG-MU)

Require: $\mathbf{V} \in \mathbb{R}_+^{F \times N}$, β, max_epoch
1: Initialise **H**, **W** with nonnegative random coefficients
2: Shuffle **V**
3: **for** $ep = 0$; $ep <$ max_epoch **do**
4: $\mathbf{b}_{rnd} \leftarrow$ permutation of $[\![1, B]\!]$
5: **for** $b \in \mathbf{b}_{rnd}$ **do**
6: Update $\mathbf{H}_b$ with (10)
7: Update **W** with (11) (with **H** replaced by $\mathbf{H}_b$)
8: **end for**
9: **end for**
10: **return H**, **W**

Figure 6: The H parameter is updated every mini-batch in both the algorithms but in cyclic updates W is updates every epoch and in ASG-MU W is updates every mini-batch

Cyclic Updates suffer from the problem that mini-batches can have similar training points together hence these points would not contribute much to learning process. That's, the reason we use Stochastic selection policy of

Algorithm 6 Greedy SG mini-batch MU rules (GSG-MU)

Require: $\mathbf{V} \in \mathbb{R}_+^{F \times N}$, β, max_epoch
Initialise $\mathbf{H}$, $\mathbf{W}$ with nonnegative random coefficients
Shuffle $\mathbf{V}$
for $ep = 0$; $ep <$ max_epoch **do**
  $\mathbf{b}_{rnd} \leftarrow$ permutation of $[\![1, B]\!]$
  for $b \in \mathbf{b}_{rnd}$ **do**
    Update $\mathbf{H}_b$ with (10)
  end for
  Update $\mathbf{W}$ with (11) (with $\mathbf{H}$ replaced by $\mathbf{H}_{[\mathbf{b}_{rnd}]_B}$)
end for
return $\mathbf{H}$, $\mathbf{W}$

Algorithm 7 Asymmetric SAG mini-batch MU rules (ASAG-MU)

Require: $\mathbf{V} \in \mathbb{R}_+^{F \times N}$, β, max_epoch
Initialise $\mathbf{H}$, $\mathbf{W}$ with nonnegative random coefficients
Shuffle $\mathbf{V}$
for $ep = 0$; $ep <$ max_epoch **do**
  $\mathbf{b}_{rnd} \leftarrow$ permutation of $[\![1, B]\!]$
  for $b \in \mathbf{b}_{rnd}$ **do**
    Update $\mathbf{H}_b$ with (10)
    Udpate $\nabla^- \mathbf{W}$ with (12)
    Udpate $\nabla^+ \mathbf{W}$ with (13)
    $\mathbf{W} \leftarrow \frac{\nabla^- \mathbf{W}}{\nabla^+ \mathbf{W}}$
  end for
end for
return $\mathbf{H}$, $\mathbf{W}$

Figure 7: Algo 6 is similar to Algo 5 we update H every mini-batch selecting the batches at random update in Algo6 we update the W matrix after one epoch only. Algo 7 we do the updating based on Stochastic Average Gradient Descent, we update both H and W at every mini-batch though

batches on the shuffled data set. $\nabla^- W_b$ refers to negative gradient negative part of the gradient of the beta divergence wrt W on the batch. Note that $\nabla_y(d_\beta(x|y)) = y^{\beta-1} - xy^{\beta-2}$ This $y^{\beta-1}$ denotes the positive gradient of the beta divergence wrt y and $xy^{\beta-2}$ denotes the negative gradient wrt y.

$$\nabla^- W \longleftarrow (1-\lambda)\nabla^- W + \lambda \nabla^-_{new} W_b \tag{13}$$

$$\nabla^+ W \longleftarrow (1-\lambda)\nabla^+ W + \lambda \nabla^+_{new} W_b \tag{14}$$

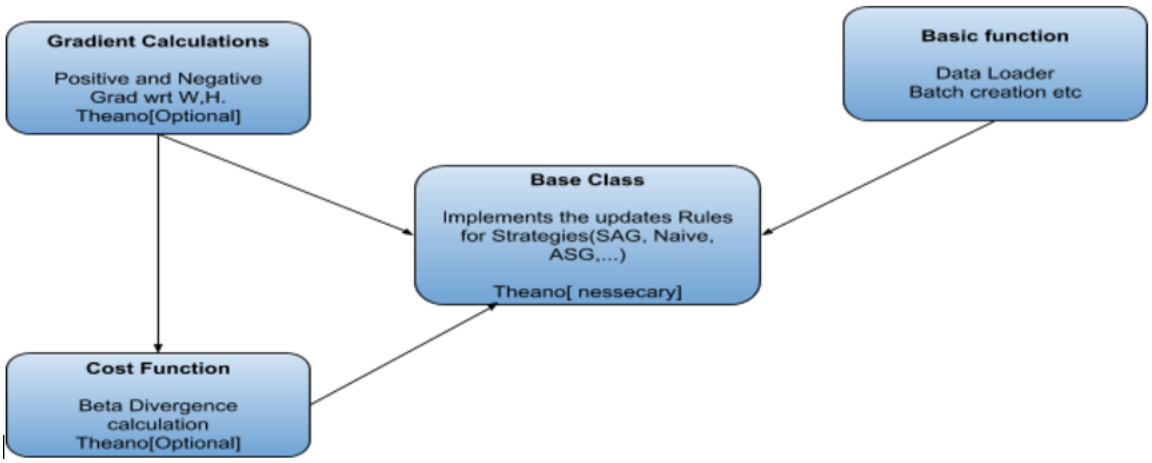

Figure 8: BetaNmF Code

4 Proposed Solution

For an online and distributive solution we combine the works of Wang et al and Kanana[8] et al. Here we describe our full solution. Where g_i indicates the ith

Initialise the matrices F ;
while *Repeat Unit data points are finished* **do**
  1.Input a chunk of data points x^t;
  2.Compute optimal $g^(t)$ by $min_g \geq 0\mathcal{L}(F, g)$;
  3. Update F;
end

Algorithm 2: Online NMF

column of G^T. The idea is very similar to HALS algorithm instead of solving 2 NLS problems as in NALS we solve N NLS problems. Point 2 can be solved by the HALS algorithm or any other standard NALS solver. Point 3 is done as follows.

$$F_{k+1}^t = P[F_k^t + 2\alpha_k \sum_1^t [x^{(s)}(g(s)^T) - F_k^{(t)} g^s (g^s)^T)]] \tag{15}$$

We use the Naive Parallel NMF Rule proposed in Kanan et al Fig2 for solving

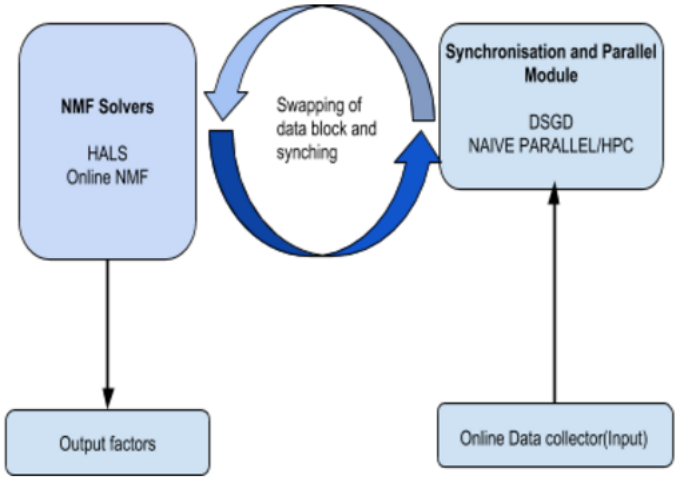

Figure 9: Proposed Architecture

the point 2. This algorithm basically distributes the rows and columns of F and G matrices into cores so that each core has only some part of each matrix. With the use of GATHERALL MPI interface, the local copy of the full factor matrix is collected at each core to perform the optimisation. This needs to looked into depth a little more for a guarantee of convergence.

4.1 Proposed API

The Api is inspired from both BetaNMF and the Sklearn NMF implmentation. We do not show here the API for the Synchronisation module. We go with sklearn implementation so that integrates with the sklearn API's seamlessly. This is no way a complete API

```
class NMF():
    """
    Non Negative matrix Factorisation Base class
    Find two non-negative matrices (W, H) whose
    product approximates the non-negative matrix X.
    This factorization can be used for example for
    dimensionality reduction, source separation
    for topic extraction.
    || X - WH ||_F^2 + alpha ||W||_1 + alpha || H||_1
    Where F indicates the forbenius norm
    || ||_1 indicates the L1 norm
    """
    def __init__(n_components, init, solver, tol, max_iter,
                 random_state):
    """
    Parameters
    ----------------------
    n_components : int or None
        Number of components,
        if n_components is not set all features
        are kept.
    init :  Method used to initialize the procedure.
        This could be one of the following
        'random' | 'nndsvd' |  'nndsvda'
        | 'nndsvdar' | 'custom'
    solver : 'cd' | 'mu' | 'musag'| 'hal' | ...
        Numerical solver to use. This could projected gradient,
        co-ordinate descent, multiplictive update,
        multiplicative updates
        with sag etc.
    tol : double, default: 1e-4
        Tolerance value used in stopping conditions.
    max_iter : integer, default: 200
        Number of iterations to compute.
    random_state : integer seed, RandomState instance,
                   or None (default)
        Random number generator seed control.
    alpha : double, default: 0.
        Regularisation constant
    shuffle : boolean, default: False
```

```
    If true, randomize the order of coordinates
    in the CD solver. Shuffle the outputs
Examples
--------
>>> import numpy as np
>>> X = np.array([[1,1], [2, 1], [3, 1.2],
                  [4, 1], [5, 0.8], [6, 1]])
>>> from gensim import NMF
>>> model = NMF(n_components=2, init='random',
    random_state=0)
>>> model.fit(X)
NMF(alpha=0.0, beta=1, eta=0.1, init='random',
    l1_ratio=0.0, max_iter=200, n_components=2,
    nls_max_iter=2000, random_state=0,
    shuffle=False, solver='cd', sparseness=None,
    tol=0.0001, verbose=0)
>>> model.components_
array([[ 2.09783018,  0.30560234],
       [ 2.13443044,  2.13171694]])
>>> model.reconstruction_err_
0.00115993...
"""
pass

def fit(self, data, **params):
"""
This module is work horse of the whole program. Here we
actually call the solvers using parameters.

Parameters
----------
X: {array-like, sparse matrix}, shape (n_samples, n_features)
        Data matrix to be decomposed
Returns
-------
self
"""
pass

def compile_theano(self):
"""
    This module compiles all theano based function
    mostly useful for multiplicative updates and FAST HALS
Returns
---------
```

```
        self
        """
        pass

    def compute_grad(self):
        """This function is responsible for computing the gradients
           we use Theano to compute gradients for us.
        Returns
        _______
        self
        """
        pass

    def transform(self, data):
        """Project data X on the basis W
        Parameters
        __________
        X : array
            The input data
        Returns
        _______
        H : array
            Activations
        """
        pass

    def prepare_batch(self, data):
        """Useful for online learning
        scheme for NMF. This module creates batches
        from the given data
        Returns
        _______________
        batch: return a subset of the data
        """
        pass
```

---

5 Deliverable

- **Primary Goals**

  Implementation of a Naive Parallel algorithm and High-Performance parallel Algorithm Defined in Kanana et al and implemented in NMLF Library.

  1. Implementation of the Online NMF solver described in the Wang et al with Naive Parallel Algorithm.

2. FAST HALS Solver implementation the Nmlf Library that uses the Naive the parallel algorithm for distribution among cores.
3. Online Multiplicative Update Solver Implementation using cython/theano/tensorfow using BaseNMF library

- **Secondary Goal**

1. Add a real live Demo example of using online NMF for text mining. Possible in recommendation systems
2. Complete the Dynamic topic Model Work PR840.

- **WishList**

1. Implementation of the synchronisation module described in LIBMF for solving the Distributed SGD.

6 Timeline

I am available for the full 3 months barring a single weekend where I am busy with some prior commitments.
I can dedicate work for long hours and give 40hrs/ week would not be challenging. I have prepared the time line so that I can finish ground work early and actually concentrate on implementing the solution in the actual GSoC period without any snags.

All dates mentioned here are weeks- starting from Monday and ending on Sunday.

4April-1May
Finish exams and other curriculum related work and projects. **4May-7May**
Introduce myself to the community and discuss at length the pros and cons of the solution proposed. **8May-14May**

- Understand the code of LibMF HALS algorithm
- Understand the Pro's and cons for implementing the proposed solution using TensorFlow and Theano
- Read the Dynamic Topic Models Blei Paper.

15May-21May

- Understand the High-Performance Parallel Nmf method from Kanan papers.
- Get comfortable with theano/tensorflow for implementing nmf code. I already have good knowledge of TensorFlow for implementing the neural network so will not be a problem
- Start Work on PR840 Get rid of all c/c++ coding styles

22May-29May

- Understand the FastHAlS implementation code the authors provide.
- Benchmark code against scikit code.
- Convert sslm class to cython PR840

22May-28May

- Finish Reading upon the Code of FAST HALS.
- Read upong BaseNMF library code for Multiplicative Updates
- Submit PR for Document Influence Model

This almost finishes PR840
29May-4June(Week 1)

- Discuss about possible implementations details of HALS with open mpi using Naive Parallel. Algorithm/HPC(whichever is picked by mentors) for now I assume we go with Naive initially.
- Finish up on MPI Readings.
- Start implementing the FastHALS code with cython/theano/tensorflow/

5June-11June(Week 2)

- Start work on Documentation.
- Finish Work on FAST HALS
- Start writing Test for FAST HALS

12June-18June(Week 3)

- Read the code for FAST HALS+ Naive Parallel Algorithm from NMF Library.
- Start implementation of FAST HALS+ Naive Parallel.
- Finish Writing Test for FAST HALS BenchMark Code against LibMF library.

19June-25 June(Week 4)

- Finish the work FastHALS(Cython) + MPI + Naive Parallel.
- Write Basic Test cases for the Fast HALS + Parallel Algorithm.
- Start to work on online Multiplicative Updates algorithm. (MU+SG)
- Milestone Reached Primary Goal 1 point 1.

26June-2 July(Week 5)

- Clean up all jobs for Mid-term evaluation.
- Write test checking the parallel implementation and Benchmarks.
- Finish Work on MU+SG algorithm
- Buffer week for Mid-term evaluation, make the work presentable.

3 July-9 July(Week 6)

- Start work on writing Blog Post. Explain the use of both Fast HALS+MU. Show real Document clustering application using the same algorithms.
- Write Test for MU+SG algorithm. Benchmark against FastHALS code.
- Start ground work on implementation of online version of NMF.(Wrt Fast HAls online implementation) This could be implementation based on Kanana et al paper [8] or going with LibMf parallel implementation [5]
- Finish online MU+SAG implementation

10-16 July(Week 7)

- Discussion on how to implement Online NMF(Show Basic Algorithm).
- Start work on online NMF implementation + integration with Naive/ HPC environment kanan et al paper.
- Start work on writing on ipynb showing online NMF

17- 23July(Week 8)

- Online NMF continues
- Start on Trivial Test for online NMF
- Finish Any remaining Documentation

24- 30July(Week 9)

- Finish Online NMF.
- Write Tests to check. parallelism and benchmark.
- Start working on MU+ SAG algorithm.

Primary Goal Achieved point 2.
Start with Secondary Goals.
31- 6August(Week 10)(2 Phase Evaluation)

- Buffer Week.
- Clean up Code finish pending works.

- Finish work MU + SAG algorithms
- Start writing tests for MU+SAG.

7- 13August(Week 11)

- Write Ipynb Demonstrating Uses of Online NMF.
- Finish Writing Documentation for online NMF.
- Finish Writing blog post discussing ONline NMF, Online Multiplicative Updates. Show benchmarks for speed against sklearn.

Primary Goal Achieved point 3.
14- 20 August(Week 12)

- Buffer Week
- Commenting Code, doc string, passing pr reviews, docker container etc

21- 27 August(Week 13) Wrap Up

- Finish Any work if remaining.
- Do cosmetic changes to the code.
- Prepare for final submission.

7 About Me

I am a pre-final year student doing Master in Computer Science from Indian Institute of Technology India. I also have a internship experience at Robert Bosch for 3 months working on Diabetic Retinopathy. I have doing machine learning for 2+ years now and have active in the open Source community for about 6 months. I have contributed in PGMPY a graphical model library in Python and also to Gensim a document modelling library in python. Some of my projects include Implementing Recommendation System using ANLS(implemented from Scratch), Library for Contextual Multi-armed Bandits(using offline evaluation), Graph Kernel based learning for sub graphs(Tentative submission at ECML-PKDD 2017), Implemented the reinforce algorithm for attention mechanism in images for generating captions paper. I have done a Deep Learning course(mostly followed Hinton Coursera), Optimisation course(Based on CMU optimisation Course). I have also done Andrew Ng Coursera course and also Daphne koller Graphical Models Course. I very motivated to apply what i have learnt till now and looking forward for great summer with Gensim.

8 Why me

I am highly motivated person for this project as I have already learnt and implemented NALS techniques in python for my Movie Recommendation Project. I have no prior commitments in the summer and no research work in the summer. Hence, I can fully concentrate on GsoC. I am already familiar with reading research journals and implementing them. I have been contributing to PGMPY for more than a 3 months now, hence I have good knowledge of contributing to a opensource project based on python. I have used gensim word2vec model extensively in my research work where I had to modify the internal implementation of the word2vec model so, I am also pretty comfortable with the gensim code base. I really Love coding and actually code for fun also. So, putting in 40 hrs + every week is not a problem for me. I have plans of pursuing a PhD in Machine Learning after completion of my masters and GsoC and in particular being associated with Gensim would give me chance to understand Machine Learning Algorithms better which in turn would be beneficial for my interviews. Finally I want to have a long term association with Gensim beyond this GsoC as I find the gensim library very helpful and easy to use.

References

[1] W. Liu, J. Wang, H. Chen, and X. Ma, "Symbolic Model Checking APSL," *2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, pp. 39–46, 2008.

[2] A. Cichocki and A. H. Phan, "Fast local algorithms for large scale nonnegative matrix and tensor factorizations," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E92-A, no. 3, pp. 708–721, 2009.

[3] C.-J. Lin, "Projected gradient methods for nonnegative matrix factorization.," *Neural computation*, vol. 19, pp. 2756–2779, 2007.

[4] C. Hsieh and I. Dhillon, "Fast coordinate descent methods with variable selection for non-negative matrix factorization," *Proceedings of the 17th ACM SIGKDD ...*, pp. 1064–1072, 2011.

[5] W.-s. Chin, Y.-c. Juan, and C.-j. Lin, "LIBMF : A Library for Parallel Matrix Factorization in Shared-memory Systems," vol. 17, pp. 1–5, 2016.

[6] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '11*, pp. 69–77, 2011.

[7] R. Serizel, S. Essid, and G. Richard, "Mini-batch stochastic approaches for accelerated multiplicative updates in nonnegative matrix factorisation with

beta-divergence," *IEEE International Workshop on Machine Learning for Signal Processing (MLSP)*, no. 3, pp. 2–7, 2016.

[8] G. Tech and H. Park, "A High-Performance Parallel Algorithm for Nonnegative Matrix Factorization," pp. 1–10, 2015.