RECOMMENDATION SYSTEM FROM FILTERING BILLIONS OF NEGATIVE DATA

By

Debkrishna Roy - 11700114028

Rajarshi Barui - 11700114047

Shubham Das - 11700114065

Tanmoy Saha - 11700114088

UNDER THE GUIDANCE OF

Mr. Koushik Mallick

PROJECT REPORT SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF BACHELOR OF TECHNOLOGY IN COMPUTER SCIENCE AND ENGINEERING

श्रमम् बिना न किमपि साध्यम्

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

RCC INSTITUTE OF INFORMATION TECHNOLOGY

[Affiliated to West Bengal University of Technology]

CANAL SOUTH ROAD, BELIAGHATA, KOLKATA-700015

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

RCC INSTITUTE OF INFORMATION TECHNOLOGY

[Affiliated to West Bengal University of Technology]

CANAL SOUTH ROAD, BELIAGHATA, KOLKATA-700015

TO WHOM IT MAY CONCERN

श्रमम् बिना न किमपि साध्यम्

I hereby recommend that the Project entitled **RECOMMENDATION SYSTEMFROM FILTERING BILLIONS OF NEGATIVE DATA** prepared under my supervision by (Reg. No. 141170110028, 141170110047, 141170110065, 141170110088 Univ. Roll No. 11700114028, 11700114047, 11700114065, 11700114088) of B.Tech. (8th Semester), may be accepted in partial fulfilment for the degree of Bachelor of Technology in Computer Science & Engineering under West Bengal University of Technology (WBUT).

PROJECT SUPERVISOR

Department of Computer Science and Engineering
RCC Institute of Information Technology

Countersigned:

HEAD OF THE DEPARTMENT

Department of Computer Sc. & Engg,
RCC Institute of Information Technology
Kolkata – 700015.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

RCC INSTITUTE OF INFORMATION TECHNOLOGY

श्रमम् बिना न किमपि साध्यम्

CERTIFICATE OF APPROVAL

The foregoing Project is hereby accepted as a credible study of an engineering subject carried out and presented in a manner satisfactory to warrant its acceptance as a prerequisite to the degree for which it has been submitted. It is understood that by this approval the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein, but approve the project only for the purpose for which it is submitted.

FINAL EXAMINATION FOR EVALUATION OF PROJECT

1. ——————————————

2. ——————————————

(Signature of Examiners)

ACKNOWLEDGEMENT

We express to our sincere gratitude to Mr. Koushik Mallick of Department of Computer Science and Engineering of RCCIIT and for extending his valuable time for us to take up his problem as a project. We would also like to thank Department of CSE, RCCIIT for providing us with the infrastructure and valuable resources as equipments and systems available in the project lab, without which we would not be able to present this project.

And lastly we would again like to thank our mentor, Mr. Koushik Mallick, without whose continued guidance, support and inspiration, we would never have been motivated to take up this project.

Debkrishna Roy

Registration Number : 141170110028

Roll Number : 11700114028

Rajarshi Barui

Registration Number : 141170110047

Roll Number : 11700114047

Shubham Das

Registration Number : 141170110065

Roll Number : 11700114065

Tanmoy Saha

Registration Number : 141170110088

Roll Number : 11700114088

Table of Contents

INTRODUCTION

The base of any recommender system relies on co-relation matrix. In normal scenario unstructured data is prepared as simple graphs. Then co-relation matrix is formed using random walk or any relevant algorithm on it.

But in any web scene there may be so many unused data nodes that hold valuable information and being ignored in the above process.

To utilize the unused data along with usable data in recommender system we need to consider multiparty graphs to prepare the data as nodes. Multiparty nodes are very inconsistent in nature and there may be thousands of nodes without any link to any other node. These nodes usually have no incoming or outgoing edge.

These nodes are generated when a user leaves ratings and comments without registering itself. Also when user deletes its account from website the user node gets as orphan node. Which translates into negative node.

Negative nodes are nothing but noise which appears to be a error incubator in process of training dataset.

Goal is to structure the multi-party graph nodes to a format by removing and reconnecting hidden edges of negative nodes to use this as input to a recommender system.

REVIEW OF LITERTURE

Recommender systems mainly work on clustering of features that end user leaves as data. The user data is used as features to train models that could cluster the available nodes into prominent clusters. But when the data sink is concerned it is often seen that end user dataset has various nodes that has no connectivity with others. These misspelled nodes are the reason behind false positive and false negative output of system.

It is a known situation in recommender system clustering that thickens the linear joint of two clusters and fit negative nodes into that space.

OBJECTIVE OF THE PROJECT

The base of any recommender system relies on co-relation matrix. In normal scenario unstructured data is prepared as simple graphs. Then co-relation matrix is formed using random walk or any relevant algorithm on it.

But in any web scene there may be so many unused data nodes that hold valuable information and being ignored in the above process.

To utilize the unused data along with usable data in recommender system we need to consider multiparty graphs to prepare the data as nodes. Multiparty nodes are very inconsistent in nature and there may be thousands of nodes without any link to any other node. These misspelled nodes are the reason behind false positive and false negative output of system.

It is a known situation in recommender system clustering that thickens the linear joint of two clusters and fit negative nodes into that space.

The main objective of this project is to remove the negative nodes that come as output in normal cluster based recommender systems.

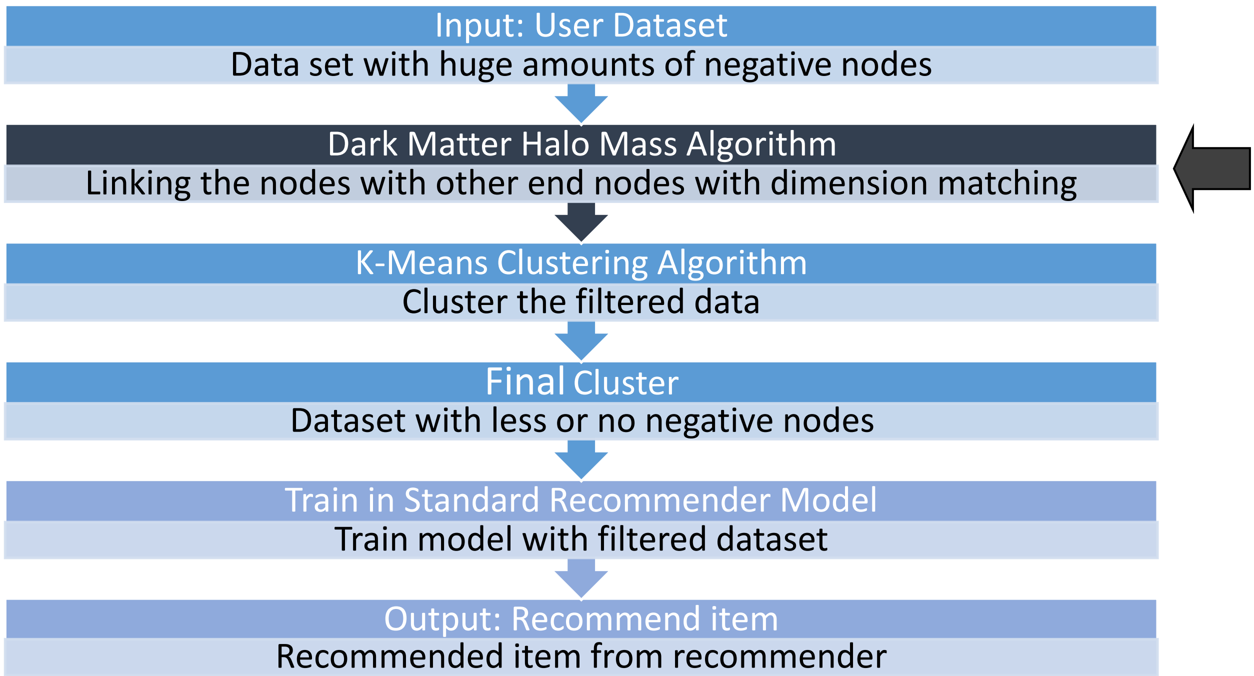
In the time of research, we found that the main problem lies in the root of text based sentiment analysis before looking for match. Any other way to achieve similarity is the main goal to overcome this issue.

We took an unusual algorithm named Dark Matter Halo Mass Function that was mainly developed for Hubble Space telescope to find similarities between light objects in darkness of the sky. We reworked the algorithm and developed a new HMFcalc function that does text mining in a different way than others.

The objective is to present a detailed overview of hmf and HMFcalc, describing its implementation and the underlying philosophy for this approach, as well as providing some worked examples that illustrate its usefulness and versatility. The paper is structured as follows. The theoretical background necessary to compute the HMF, setting out a compilation of HMF fitting functions drawn from the literature and demonstrating how the HMF differs in CDM and WDM models. Describe our implementation of hmf and HMFcalc and discuss the algorithms and methods used and present some worked examples using HMFcalc.

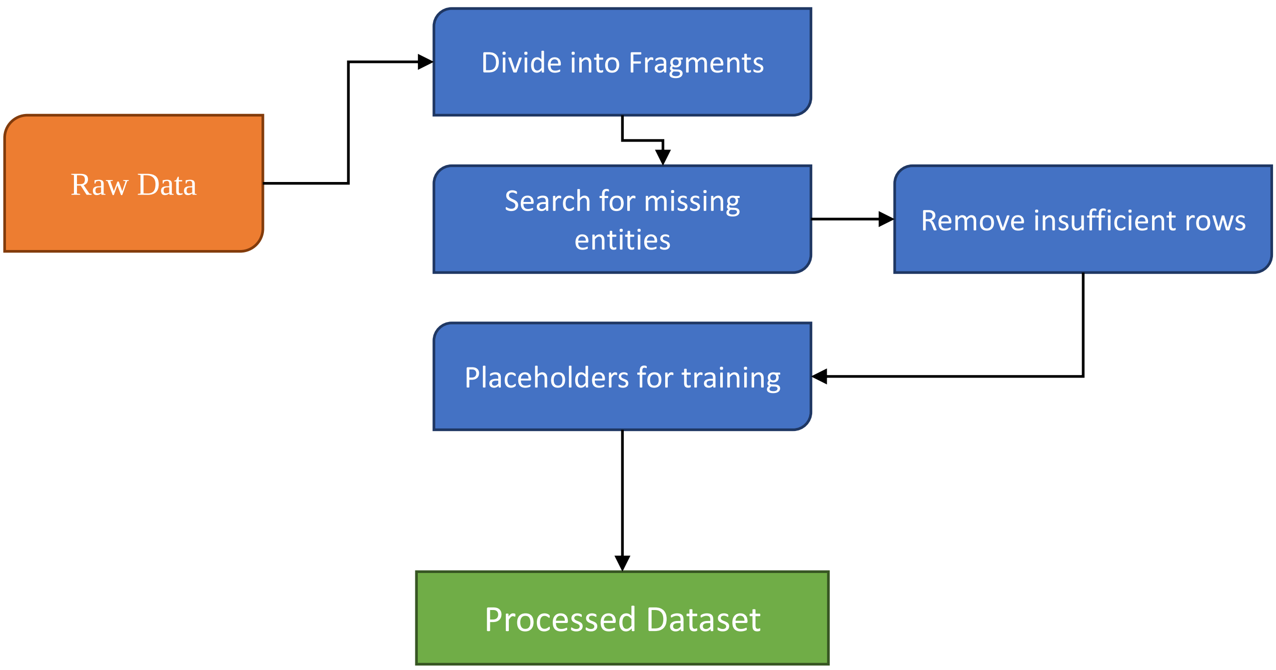
SYSTEM DESIGN

LOGIC FLOW:

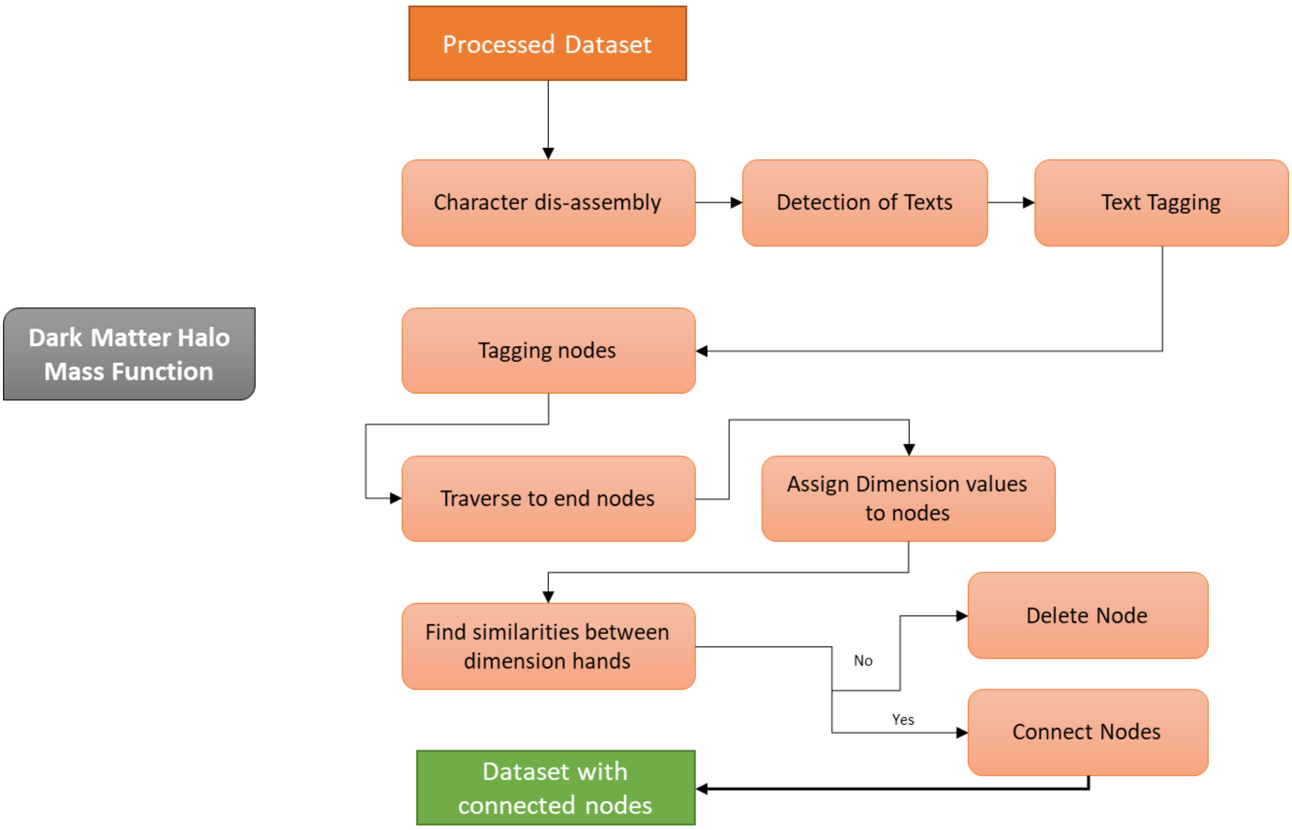


BLOCK FLOW:

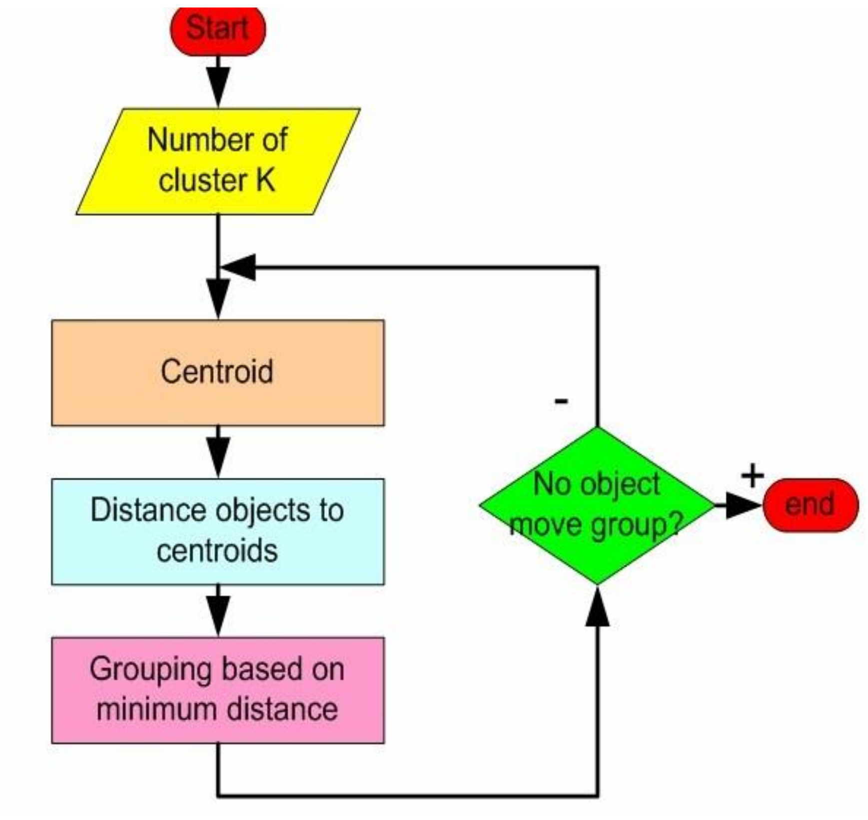
Data pre-processing



Removal of Negative nodes:



Clustering:



K-Means Diagram: Collected from Internet

Methodology

An important problem that arises when we search for similar items of any kind is that there may be far too many pairs of items to test each pair for their degree of similarity, even if computing the similarity of any one pair can be made very easy. Only text based systems usually try for capturing sentiments where the similarity might vary from original when texts are tested against NLP.

The other way around is to assign singular nodes multiple dimension hands with specific codes sourced from common set. So that measuring similarity is bounded within texts and distances but not on sentiments.

Dark Matter Halo Mass Algorithm

Halo mass function is the main algorithm used in Hubble Space Telescope to recognise link between light sources found in dark sky.

The dark matter halo mass function (HMF) is a characteristic property of cosmological structure formation models, quantifying the number density of dark matter haloes per unit mass in the Universe. A key goal of current and planned large galaxy surveys is to measure the HMF and to use it to test theories of dark matter and dark energy. We present a new web application for calculating the HMF – the frontend HMFcalc and the engine hmf. HMFcalc has been designed to be flexible, efficient and easy to use, providing observational and theoretical astronomers alike with the means to explore standard functional forms of the HMF or to tailor their own. We outline the theoretical background needed to compute the HMF, we show how it has been implemented in hmf, and finally we provide worked examples that illustrate HMFcalc's versatility as an analysis tool.

We have used the cosmological structure formation model to derive features and to tag them with multiple dimension hands.

Brief of the original algorithm:

A wealth of compelling observational evidence in a Universe whose matter content is predominantly dark („84%; cf. Ade et al., 2013) and nonbaryonic in nature (cf. Bergstr¨om, 2000). Theories of cosmological structure formation predict that dark matter clusters into massive gravitationally bound structures called haloes. The dark matter halo mass function (hereafter HMF) quantifies the number of these haloes per unit comoving volume of the Universe as a function of their mass. The HMF is sensitive to the cosmological parameters, primarily the mass-energy density of dark matter Ωc and dark energy ΩΛ (e.g. Murray et al., 2013), but it also depends on the nature of the dark matter. The standard Cold Dark Matter (CDM) model predicts an HMF in which the number of haloes increases with decreasing halo mass M approximately as M´1.8 (e.g. Luki´c et al., 2007; Bhattacharya et al., 2011), whereas viable Warm Dark Matter (WDM) models predict fewer haloes than the CDM model at low masses(e.g. Schneider et al., 2013; Pacucci et al., 2013). The potential of the HMF as a probe of dark matter and dark energy is widely recognised (e.g. Tinker and Kravtsov, 2008; Vikhlinin et al., 2009) and is one of the key science drivers of current and planned future galaxy surveys (Driver, 2011; Pierre et al., 2011). Cosmological N-body simulations are now established as the tool for studying the HMF (cf. the recent review by Knebe et al., 2013), but the information contained in a simulation is usually distilled and recast in a more compact form. Usually this is the comoving number density of haloes per unit logarithm of the halo mass M,

$$\frac{dn}{d\ln M} = M \cdot \frac{\rho_0}{M^2} f(\sigma) \left| \frac{d\ln\sigma}{d\ln M} \right|;$$

here σ and ρ0 are the cosmology-dependent mass variance and mean density and fpσq represents the functional form that defines a particular HMF fit. Eq 1 is not difficult to compute, but neither is it straightforward.

A function is prepared on this context to measure temperature index of text data named HMFcalc.

HMFcalc can be used in a number of ways, including as • a standard against which to check one's own code; • an easy-to-use interface to generate HMFs against which to check observational/simulations data; and • a visually intuitive way to explore the effects of cosmology on the HMF.

The objective is to present a detailed overview of hmf and HMFcalc, describing its implementation and the underlying philosophy for this approach, as well as providing some worked examples that illustrate its usefulness and versatility. The paper is structured as follows. The theoretical background necessary to compute the HMF, setting out a compilation of HMF fitting functions drawn from the literature and demonstrating how the HMF differs in CDM and WDM models. Describe our implementation of hmf and HMFcalc and discuss the algorithms and methods used and present some worked examples using HMFcalc.

The Halo mass Function:

The HMF quantifies the number of dark matter haloes per unit mass per unit by calculating volume of the Universal light spectrum range.

$$\frac{dn}{d\ln M} = M \cdot \frac{\rho_0}{M^2} f(\sigma) \left| \frac{d\ln\sigma}{d\ln M} \right|$$

where fpσq is the fitting function that we shall return to shortly, ρ0 is the mean density of the Universal light spectrum and σ is the rms variance of mass within a sphere of radius R that contains mass M,

$$M = \frac{4\pi\rho_0}{3} R^3.$$

Mass variance is calculated via an integral

$$\sigma^2(R) = \frac{1}{2\pi^2} \int_0^\infty k^2 P(k) W^2(kR) dk$$

The right-most factor of Equation can be written as

$$\frac{d\ln\sigma}{d\ln M} = \frac{3}{2\sigma^2\pi^2R^4} \int_0^\infty \frac{dW^2(kR)}{dM} \frac{P(k)}{k^2} dk$$

The window function and its derivative are functions of the product kR, but we evaluate Eqs 3 and 5 by integrating over k. For this reason, care must be taken when solving the integrals numerically to ensure that the results are converged. We demonstrate why in Fig 1, where we plot şkR 0 W2pxqdx andş8 kR W2pxqdx. The integralşkR 0 W2pxqdx allows us to identify an upper limit on the minimum kR required for convergence; we want the range of kR for any R to have a minimum that bounds the non-zero parts of the function.

The primordial power spectrum, imprinted during the epoch of inflation during the first moments after the Big Bang, is expected to have a form Ppkq9kn. The transfer function quantifies how this primordial form is modified on different scales, and it is particularly sensitive to the nature of the dark matter and the baryon density parameter Ωb. We use the public Code for Anisotropies in the

Microwave Background (CAMB) (Lewis et al., 2000) to compute our transfer functions.

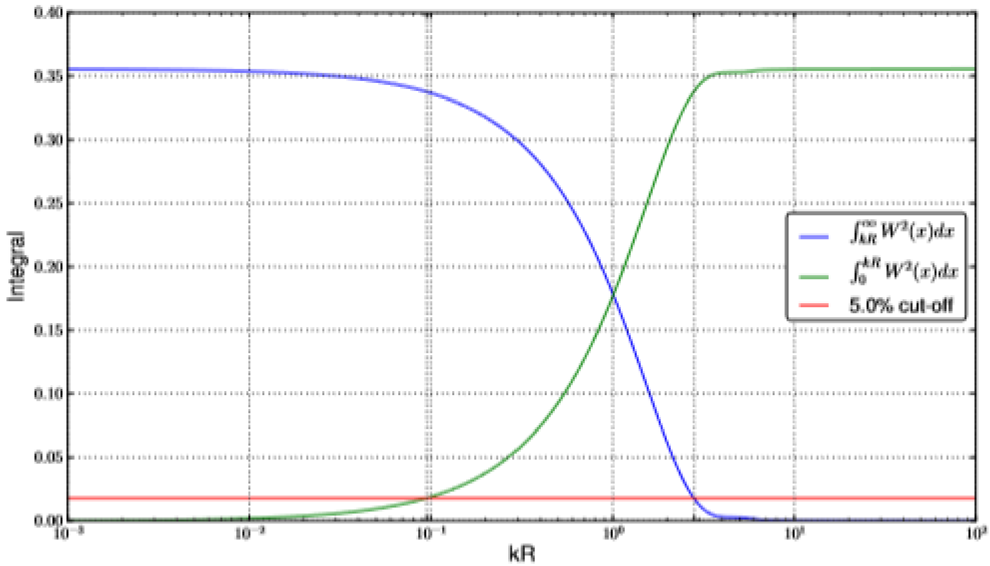


Fig: HMF error vs time (generated in Jupyter using pyplot)

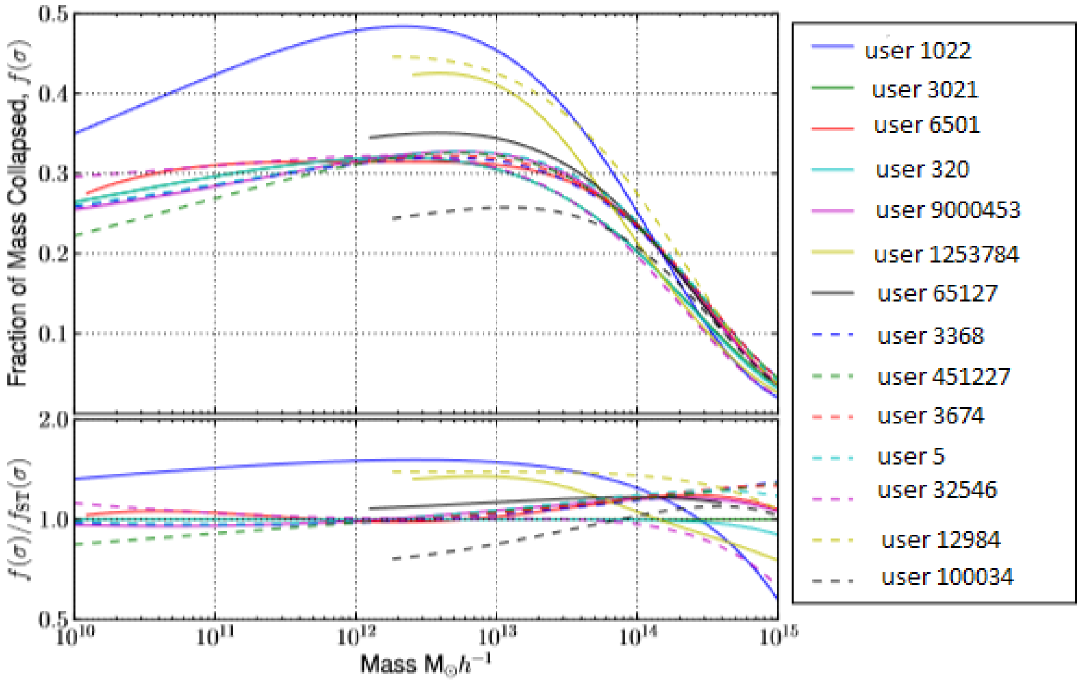
HMF Fitting:

Fitting algorithm works based upon a previous work by Press and Schechter (1974) (hereafter PS) and Bond et al. (1991) established a simple form for fpσq by assuming that haloes form by spherical collapse, finding

$$f(\sigma) = \sqrt{\frac{2}{\pi}} \frac{\delta_c}{\sigma} \exp\left(-\frac{\delta_c^2}{2\sigma^2}\right),$$

where δc » 1.686 is the critical overdensity for spherical collapse. However, N-body simulations of cosmological structure formation have revealed that the PS form underestimates the abundance of higher mass haloes and overestimates the abundance of lower mass haloes. (e.g. Sheth et al., 2001; White, 2002; Luki´c et al., 2007). Sheth et al. (2001) (hereafter ST) explored an extension to the PS formalism by considering ellipsoidal rather than spherical collapse and obtained a form for the mass function that is identical to Eq 1 but with a modified fpσq. Subsequent studies have largely adopted the same philosophical approach of assuming that the HMF can be expressed in the form of Eq 1 and using fpσq to characterise the HMF. Table 1 provides a concise summary of the forms for fpσqthat have appeared in the literature to date and which are included in

HMFcalc, and we list also the cosmology and mass and redshift ranges over which the fits have been made.



All fitting functions at redshift zero over a large mass range (limits placed as appropriate on each function). Lower: each fitting function divided by the Sheth-Tormen fit.

Warm-Dak Matter Fit:

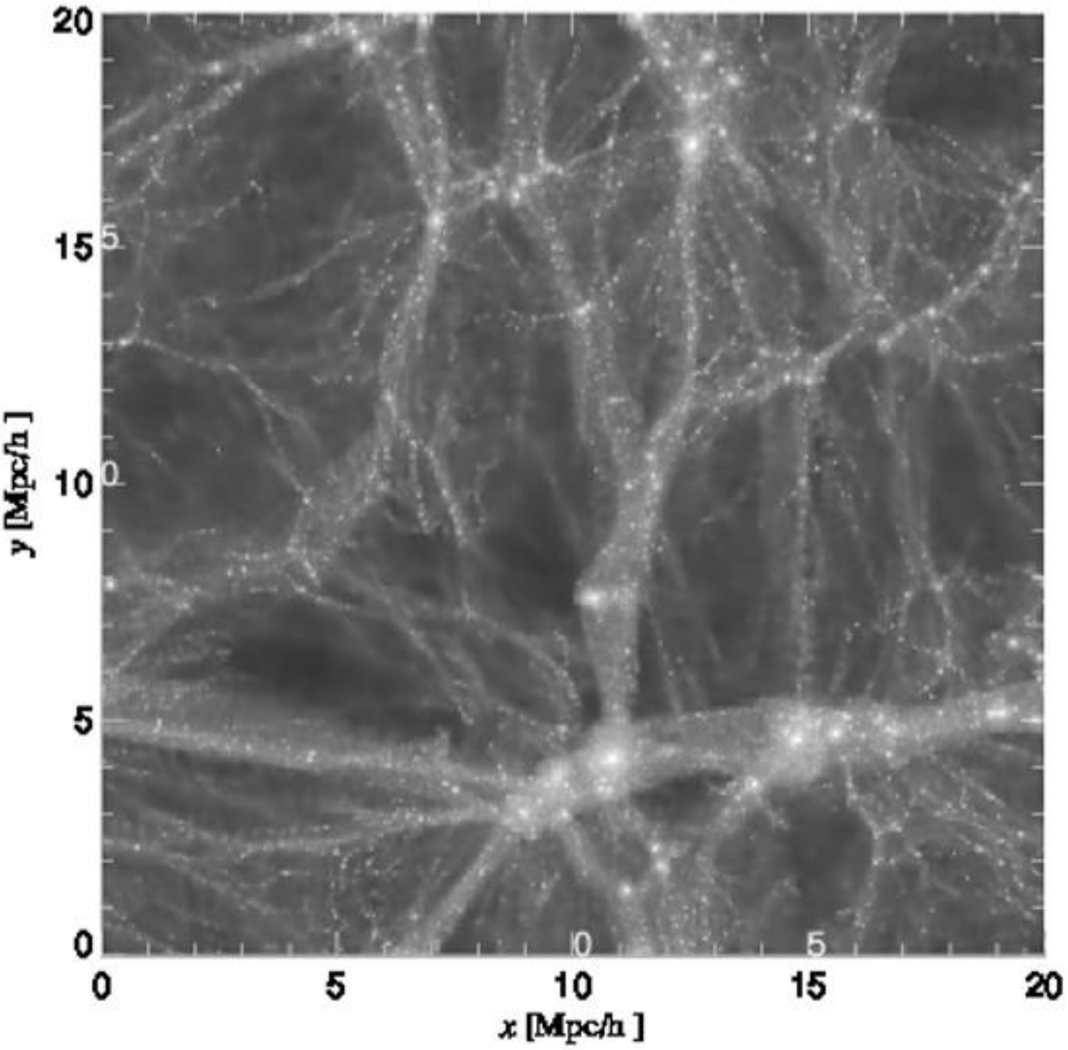
$$T_k^X = \left(1 + (\alpha k)^{2\nu}\right)^{-5/\nu},$$

With v=1.2

$$\alpha = 0.048 \left(\frac{\Omega_X}{0.4}\right)^{.15} \left(\frac{h}{.65}\right)^{1/3} \left(\frac{1}{m_X}\right)^{1.15} \left(\frac{1.5}{g_X}\right)^{.29}$$

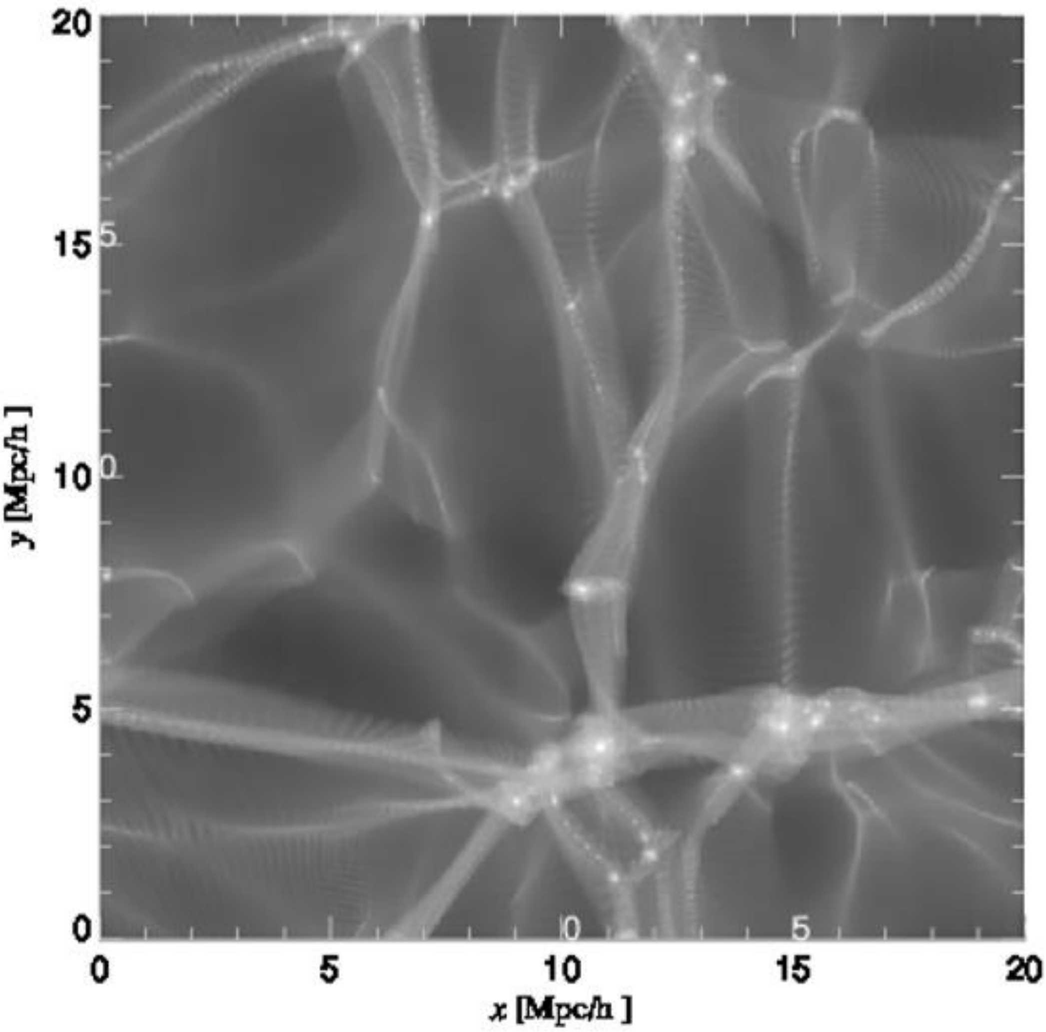
with ΩX the current fractional density of the WDM particle (this can be taken as equivalent to the CDM density Ωcdm in a single-species WDM model), mX is the particle mass in keV, and gX controls the abundance of the species relative to photons and has the fiducial value of 1.5 for a light neutrino. By default in HMFcalc, we assume that ν and gX are set to their fiducial values and allow only a single-species model; the only free parameter that we allow is mx.

Visual Impression:



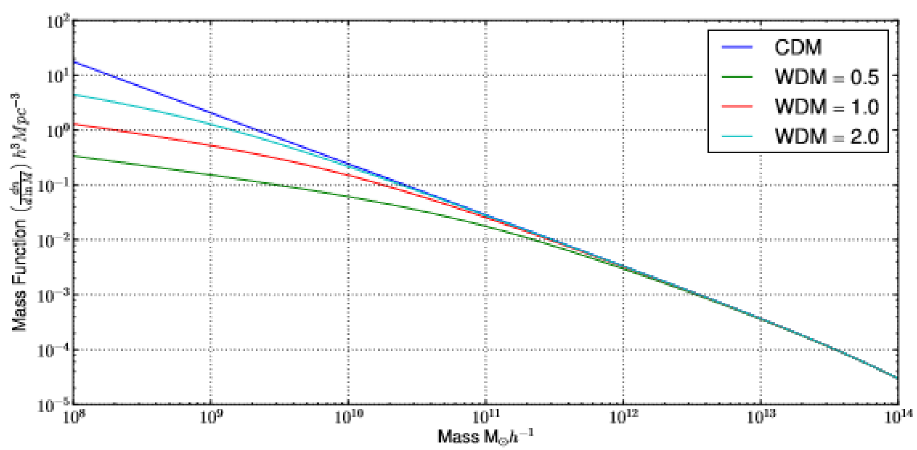
An image fragment of 0.5% of the original captured image by telescope. Visual impression of the projected dark matter density in a cosmological N-body simulations of a 20 h´1Mpc box, modelling the growth of structure in a fiducial CDM model (left panel) and its WDM counterpart (right panel). For the WDM model we assume a particle mass of mX=0.5 keV/c2. Note the absence of small-scale structure (i.e. low mass dark matter haloes) in the WDM run compared to the CDM run.

After ligh-spectrum is taken as unique column of tag elements and run through a dimension-hand orientation process the caputer image seemed better classified and it depicts real origin and negative pixels are removed.



In the output it can clearly observed that original light noise was cancelled out.

Here are the threshold batch test bench score:



K-means Clustering

K-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. *k*-means clustering aims to partition *n* observations into *k* clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster.

The algorithm has a loose relationship to the *k*-nearest neighbor classifier, a popular machine learning technique for classification that is often confused with *k*-means because of the *k* in the name. One can apply the 1-nearest neighbor classifier on the cluster centers obtained by *k*-means to classify new data into the existing clusters. This is known as nearest centroid classifier.

Distance Measures:

Common distance measures include the Euclidean distance, the Euclidean squared distance and the Manhattan or City distance.

The Euclidean measure corresponds to the shortest geometric distance between two points.

$$d = \sqrt{\sum_{i=1}^{N}(x_i - y_i)^2}$$

A faster way of determining the distance is by use of the squared Euclidean distance which calculates the above distance squared, i.e.

$$d_{sq} = \sum_{i=1}^{N}(x_i - y_i)^2$$

The Manhattan measure calculates a distance between points based on a grid and is illustrated in Figure 1.1.

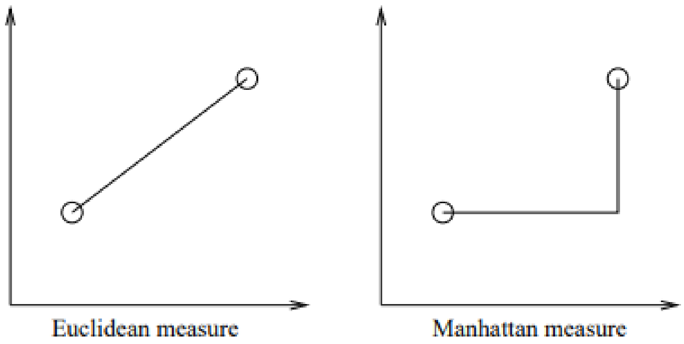


Figure 1.1: Comparision between the Euclidean and the Manhattan measure.

The following figures illustrate the K-means algorithm on a 2-dimensional data set.

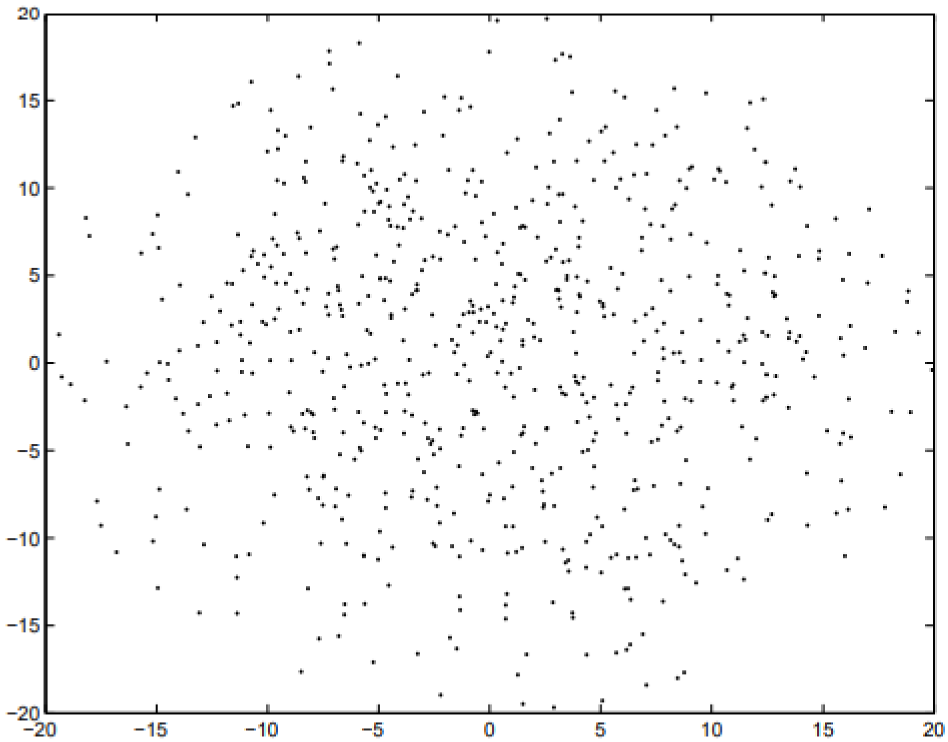


Figure 1.2: Example of signal data made from Gaussian White Noise.

Here the nodes are scattered in a space. Target is to cluster them according to features.

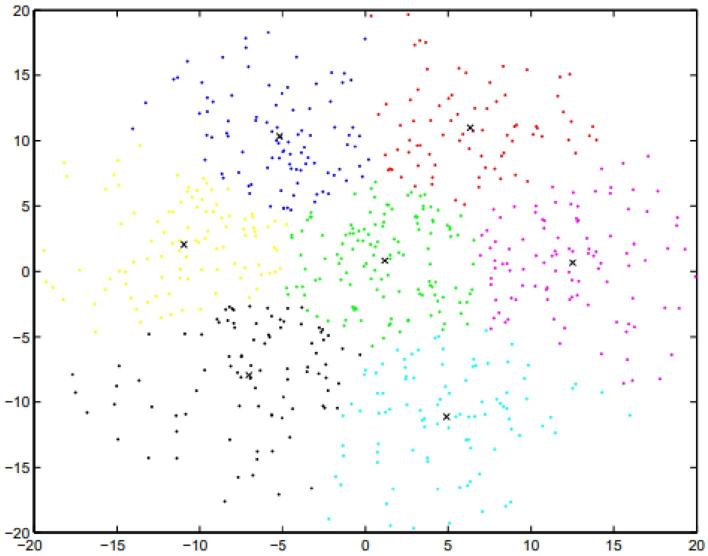


Figure 1.3: The signal data are seperated into seven clusters. The centroids are marked with a cross.

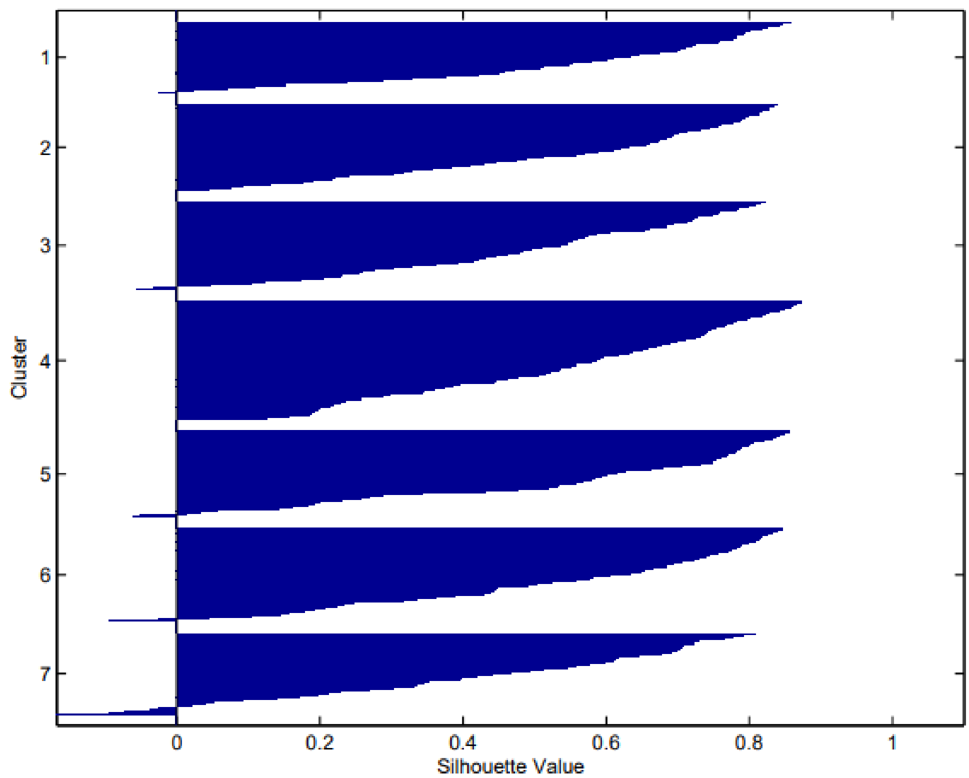


Figure 1.4: The Silhouette diagram shows how well the data are seperated into the seven clusters. If the distance from one point to two centroids is the same, it means the point could belong to both centroids. The result is a conflict which gives a negative value in the Silhouette diagram. The positive part of the Silhuoette diagram, shows that there is a clear seperation of the points between the clusters.

K-Means Usage:

The main intuition behind our implementation is as follows.

All the nodes are potential candidates for the closest node at the root level. However, for the children of the root node, we may be able to prune the candidate set by using simple geometrical constraints. Clearly, each child node will literally have different candidate sets. Further, a given prototype may belong to the candidate set of several child nodes. This approach can be applied recursively till the size of the candidate set is one for each node. At this stage, all the patterns in the subspace represented by the subtree have the sole candidate as their closest prototype. Using this approach, we expect that the number of distance calculation for the first loop (in Figure 1) will be proportional to $n \times F(k, d)$ where $F(k, d)$ where $F(k, d)$ is much smaller than $f(k, d)$. This is because the distance calculation has to be performed only with internal nodes (representing many patterns) and not the patterns themselves in most cases. This approach can also be used to significantly reduce the time requirements for calculating the prototypes for the next iteration (second for loop in Figure 1). We also expect the time requirement for the second for loop to be proportional to $n \times F(k, d)$.

The improvements obtained using our approach are crucially dependent on obtaining good pruning methods for obtaining candidate sets for the next level.

The above strategy guarantees that no candidate is pruned if it can potentially be closer than any other candidate prototype to a given subspace. Our algorithm is based on organizing the pattern vectors so that one can find all the patterns which are closest to a given prototype efficiently. In the first phase of the algorithm, we build a k-d tree to organize the pattern vectors. The root of such a tree represents all the patterns, while the children of the root represent subsets of the patterns completely contained in subspaces (Boxes). The nodes at the lower levels represent smaller boxes. For building the k-d tree, there are several competing choices which affect the overall structure.

1. Choice of dimension used for performing the split: One option is to choose a common dimension across all the nodes at the same level of the tree. The dimensions are chosen in a round-robin fashion for different levels as we go down the tree. The second option is to use the splitting dimension with the longest length.
2. Choice of splitting point along the chosen dimension: We tried two approaches based on choosing the central splitting point or median splitting point. The former divides the splitting dimensions into two equal

parts (by width) while the latter divides the dimensions such that there are equal number of patterns on either side. We will refer to these approaches as midpoint-based and median-based approaches respectively. Clearly, the cost of the median-based approach is slightly higher as it requires calculation of the median.

Algorithm in use:

function Direct-k-means()

Initialize k prototypes $(w_1, \ldots, w_k)$ such that $w_j = i_l,\ j \in \{1, \ldots, k\},\ l \in \{1, \ldots, n\}$

Each cluster C_j is associated with prototype w_j

Repeat

for each input vector i_l, where $l \in \{1, \ldots, n\}$, *do*

Assign i_l to the cluster C_{j*} with nearest prototype w_{j*} (i.e., $\mid i_l - w_{j*} \mid \ \leq \ \mid i_l - w_j \mid,\ j \in \{1, \ldots, k\}$)

for each cluster C_j, where $j \in \{1, \ldots, k\}$, *do*

Update the prototype w_j to be the centroid of all samples currently in C_j, so that $w_j = \sum_{i_l \in C_j} i_l / \mid C_j \mid$

Compute the error function:

$$E = \sum_{j=1}^{k} \sum_{i_l \in C_j} \mid i_l - w_j \mid^2$$

Until E does not change significantly or cluster membership no longer changes *function TraverseTree*($node$, $\overline{p}$,l,d)

$Alive$ = Pruning($node$, $\overline{p}$,l,d)

if | $Alive$ |= 1 *then*

/* All the points in $node$ belong to the alive cluster */

Update the centroid's statistics based on the information stored in the node

return

if $node$ is a leaf *then*

for each point in $node$

Find the nearest prototype p_i

Assign point to p_i

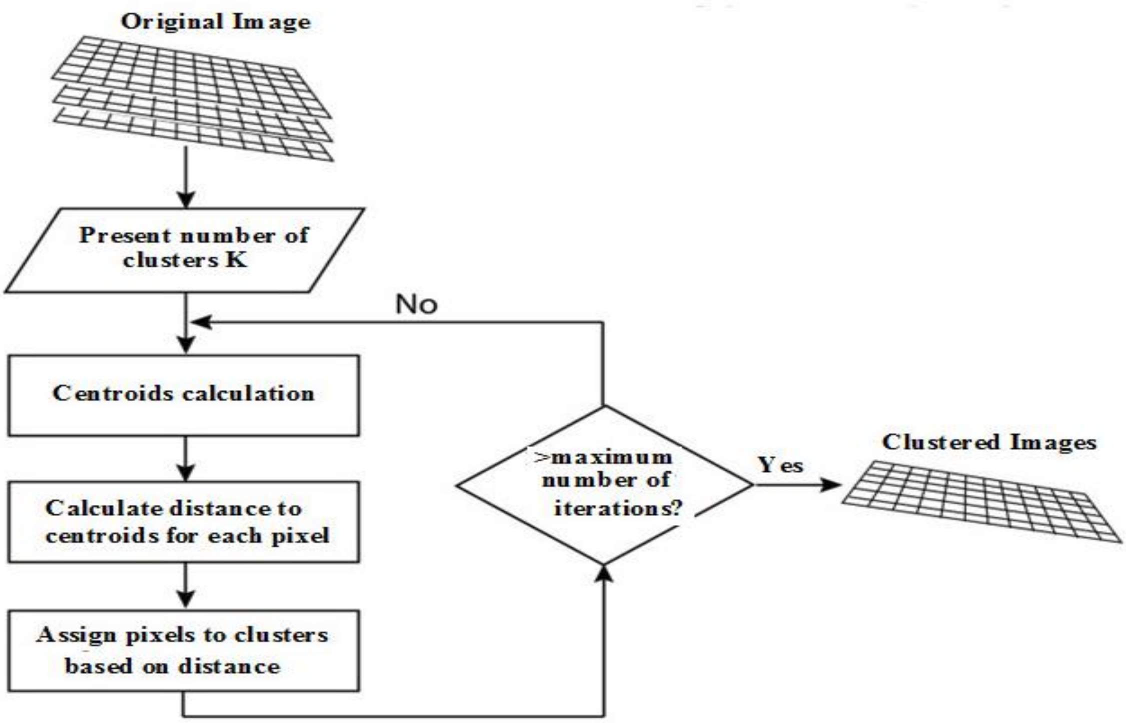
Update the centroid's statistics

return

for each *child* node *do*

TraverseTree($child$,$Alive$,| $Alive$ |,d)

Basic Flowchart of K-means Standard Algorithm:



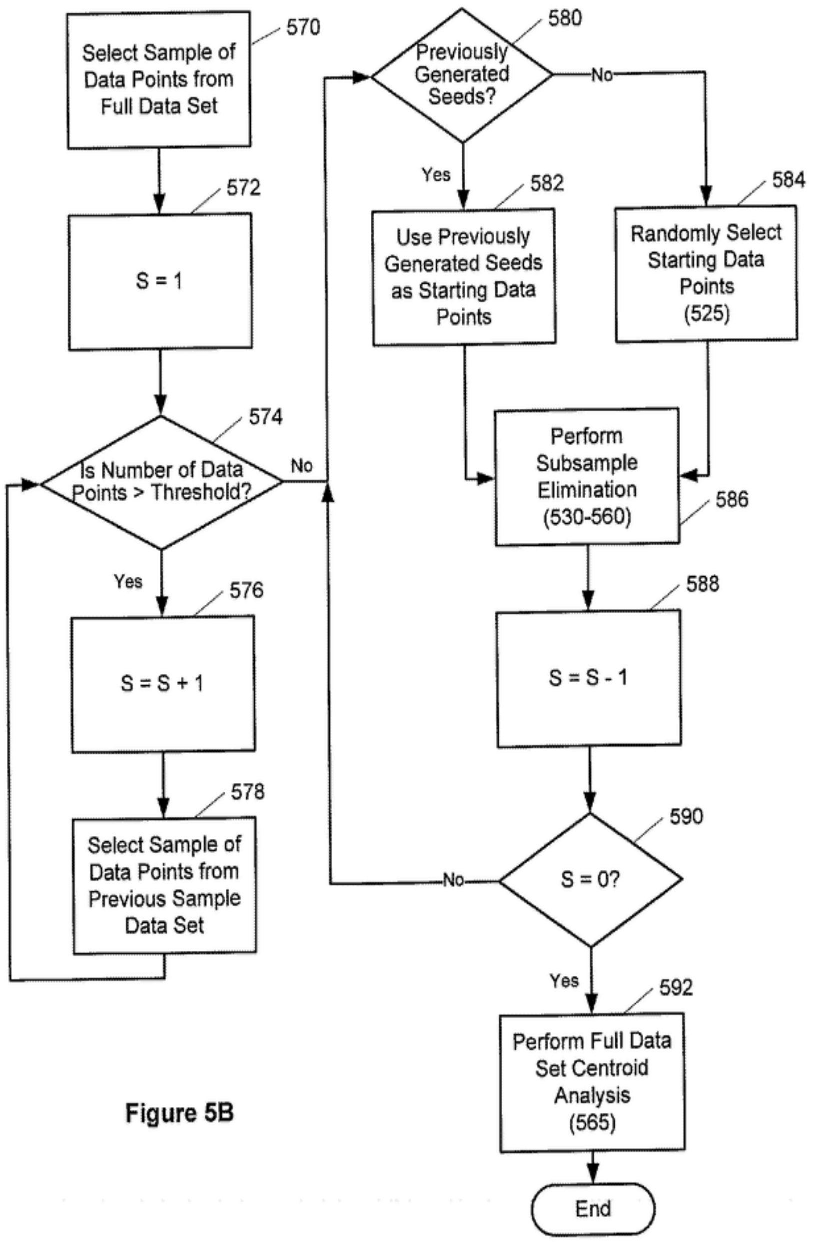
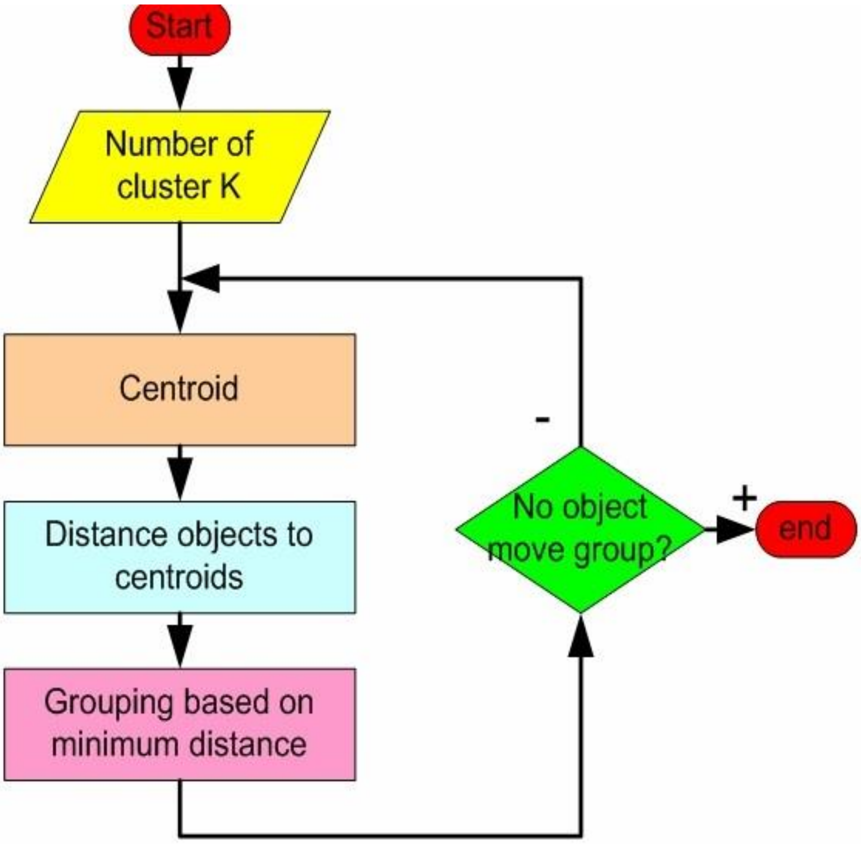

570
Select Sample of Data Points from Full Data Set
572
S = 1
574
Is Number of Data Points > Threshold?
No
Yes
576
S = S + 1
578
Select Sample of Data Points from Previous Sample Data Set
580
Previously Generated Seeds?
No
Yes
582
Use Previously Generated Seeds as Starting Data Points
584
Randomly Select Starting Data Points (525)
Perform Subsample Elimination (530-560)
586
588
S = S - 1
590
S = 0?
No
Yes
592
Perform Full Data Set Centroid Analysis (565)
End

Figure 5B

Basic Diagram of K-Means:



Implementation

K-Means code:

```
import math
import random

plotly = False
try:
    import plotly
    from plotly.graph_objs import Scatter, Scatter3d, Layout
except ImportError:
    print "INFO: Plotly is not installed, plots will not be
generated."

def main():

    # How many points are in our dataset?
    num_points = 20

    # For each of those points how many dimensions do they
have?
    # Note: Plotting will only work in two or three
dimensions
    dimensions = 2

    # Bounds for the values of those points in each dimension
    lower = 0
    upper = 200

    # The K in k-means. How many clusters do we assume exist?
    num_clusters = 3

    # When do we say the optimization has 'converged' and
stop updating clusters
    cutoff = 0.2

    # Generate some points to cluster
    points = [
        makeRandomPoint(dimensions, lower, upper) for i in
xrange(num_points)
    ]

    # Cluster those data!
    clusters = kmeans(points, num_clusters, cutoff)

    # Print our clusters
    for i, c in enumerate(clusters):
        for p in c.points:
            print " Cluster: ", i, "\t Point :", p

    # Display clusters using plotly for 2d data
    if dimensions in [2, 3] and plotly:
        print "Plotting points, launching browser ..."
        plotClusters(clusters, dimensions)

class Point(object):
    '''
    A point in n dimensional space
```

```
    '''
    def __init__(self, coords):
        '''
        coords - A list of values, one per dimension
        '''

        self.coords = coords
        self.n = len(coords)

    def __repr__(self):
        return str(self.coords)

class Cluster(object):
    '''
    A set of points and their centroid
    '''

    def __init__(self, points):
        '''
        points - A list of point objects
        '''

        if len(points) == 0:
            raise Exception("ERROR: empty cluster")

        # The points that belong to this cluster
        self.points = points

        # The dimensionality of the points in this cluster
        self.n = points[0].n

        # Assert that all points are of the same
dimensionality
        for p in points:
            if p.n != self.n:
                raise Exception("ERROR: inconsistent
dimensions")

        # Set up the initial centroid (this is usually based
off one point)
        self.centroid = self.calculateCentroid()

    def __repr__(self):
        '''
        String representation of this object
        '''
        return str(self.points)

    def update(self, points):
        '''
        Returns the distance between the previous centroid
and the new after
        recalculating and storing the new centroid.

        Note: Initially we expect centroids to shift around a
lot and then
        gradually settle down.
        '''
        old_centroid = self.centroid
        self.points = points
        self.centroid = self.calculateCentroid()
```

```
        shift = getDistance(old_centroid, self.centroid)
        return shift

    def calculateCentroid(self):
        '''
        Finds a virtual center point for a group of n-
dimensional points
        '''
        numPoints = len(self.points)
        # Get a list of all coordinates in this cluster
        coords = [p.coords for p in self.points]
        # Reformat that so all x's are together, all y'z etc.
        unzipped = zip(*coords)
        # Calculate the mean for each dimension
        centroid_coords = [math.fsum(dList)/numPoints for
dList in unzipped]

        return Point(centroid_coords)

def kmeans(points, k, cutoff):

    # Pick out k random points to use as our initial
centroids
    initial = random.sample(points, k)

    # Create k clusters using those centroids
    # Note: Cluster takes lists, so we wrap each point in a
list here.
    clusters = [Cluster([p]) for p in initial]

    # Loop through the dataset until the clusters stabilize
    loopCounter = 0
    while True:
        # Create a list of lists to hold the points in each
cluster
        lists = [[] for _ in clusters]
        clusterCount = len(clusters)

        # Start counting loops
        loopCounter += 1
        # For every point in the dataset ...
        for p in points:
            # Get the distance between that point and the
centroid of the first
            # cluster.
            smallest_distance = getDistance(p,
clusters[0].centroid)

            # Set the cluster this point belongs to
            clusterIndex = 0

            # For the remainder of the clusters ...
            for i in range(clusterCount - 1):
                # calculate the distance of that point to
each other cluster's
                # centroid.
                distance = getDistance(p,
clusters[i+1].centroid)
                # If it's closer to that cluster's centroid
```

```
update what we
                # think the smallest distance is
                if distance < smallest_distance:
                    smallest_distance = distance
                    clusterIndex = i+1
            # After finding the cluster the smallest distance
away
            # set the point to belong to that cluster
            lists[clusterIndex].append(p)

        # Set our biggest_shift to zero for this iteration
        biggest_shift = 0.0

        # For each cluster ...
        for i in range(clusterCount):
            # Calculate how far the centroid moved in this
iteration
            shift = clusters[i].update(lists[i])
            # Keep track of the largest move from all cluster
centroid updates
            biggest_shift = max(biggest_shift, shift)

        # If the centroids have stopped moving much, say
we're done!
        if biggest_shift < cutoff:
            print "Converged after %s iterations" %
loopCounter
            break
    return clusters

def getDistance(a, b):
    '''
    Euclidean distance between two n-dimensional points.

https://en.wikipedia.org/wiki/Euclidean_distance#n_dimensions
    Note: This can be very slow and does not scale well
    '''
    if a.n != b.n:
        raise Exception("ERROR: non comparable points")

    accumulatedDifference = 0.0
    for i in range(a.n):
        squareDifference = pow((a.coords[i]-b.coords[i]), 2)
        accumulatedDifference += squareDifference
    distance = math.sqrt(accumulatedDifference)

    return distance

def makeRandomPoint(n, lower, upper):
    '''
    Returns a Point object with n dimensions and values
between lower and
    upper in each of those dimensions
    '''
    p = Point([random.uniform(lower, upper) for _ in
range(n)])
    return p

def plotClusters(data, dimensions):
```

```
    '''
    This uses the plotly offline mode to create a local HTML
file.
    This should open your default web browser.
    '''
    if dimensions not in [2, 3]:
        raise Exception("Plots are only available for 2 and 3
dimensional data")

    # Convert data into plotly format.
    traceList = []
    for i, c in enumerate(data):
        # Get a list of x,y coordinates for the points in
this cluster.
        cluster_data = []
        for point in c.points:
            cluster_data.append(point.coords)

        trace = {}
        centroid = {}
        if dimensions == 2:
            # Convert our list of x,y's into an x list and a
y list.
            trace['x'], trace['y'] = zip(*cluster_data)
            trace['mode'] = 'markers'
            trace['marker'] = {}
            trace['marker']['symbol'] = i
            trace['marker']['size'] = 12
            trace['name'] = "Cluster " + str(i)
            traceList.append(Scatter(**trace))
            # Centroid (A trace of length 1)
            centroid['x'] = [c.centroid.coords[0]]
            centroid['y'] = [c.centroid.coords[1]]
            centroid['mode'] = 'markers'
            centroid['marker'] = {}
            centroid['marker']['symbol'] = i
            centroid['marker']['color'] = 'rgb(200,10,10)'
            centroid['name'] = "Centroid " + str(i)
            traceList.append(Scatter(**centroid))
        else:
            symbols = [
                "circle",
                "square",
                "diamond",
                "circle-open",
                "square-open",
                "diamond-open",
                "cross", "x"
            ]
            symbol_count = len(symbols)
            if i > symbol_count:
                print "Warning: Not enough marker symbols to
go around"
            # Convert our list of x,y,z's separate lists.
            trace['x'], trace['y'], trace['z'] =
zip(*cluster_data)
```

```
                trace['mode'] = 'markers'
                trace['marker'] = {}
                trace['marker']['symbol'] = symbols[i]
                trace['marker']['size'] = 12
                trace['name'] = "Cluster " + str(i)
                traceList.append(Scatter3d(**trace))
                # Centroid (A trace of length 1)
                centroid['x'] = [c.centroid.coords[0]]
                centroid['y'] = [c.centroid.coords[1]]
                centroid['z'] = [c.centroid.coords[2]]
                centroid['mode'] = 'markers'
                centroid['marker'] = {}
                centroid['marker']['symbol'] = symbols[i]
                centroid['marker']['color'] = 'rgb(200,10,10)'
                centroid['name'] = "Centroid " + str(i)
                traceList.append(Scatter3d(**centroid))

    title = "K-means clustering with %s clusters" %
str(len(data))
    plotly.offline.plot({
        "data": traceList,
        "layout": Layout(title=title)
    })

if __name__ == "__main__":
    main()
```

Halo Mass Function Codes:

Attributes.py

attribute.py

```python

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function


class Attributes(object):
  def __init__(self, num_feature_cat=0, feature_cat=None,
               num_text_feat=0, feature_mulhot=None, mulhot_max_length=None,
               mulhot_starts=None, mulhot_lengths=None,
               v_sizes_cat=None, v_sizes_mulhot=None,
               embedding_size_list_cat=None):
    self.num_features_cat = num_feature_cat
    self.num_features_mulhot = num_text_feat
    self.features_cat = feature_cat
    self.features_mulhot = feature_mulhot
    # self.mulhot_max_length = mulhot_max_length
    self.mulhot_starts = mulhot_starts
    self.mulhot_lengths = mulhot_lengths
    self._embedding_classes_list_cat = v_sizes_cat
    self._embedding_classes_list_mulhot = v_sizes_mulhot
    return

  def set_model_size(self, sizes, opt=0):
    if isinstance(sizes, list):
      if opt == 0:
        assert(len(sizes) == self.num_features_cat)
        self._embedding_size_list_cat = sizes
      else:
        assert(len(sizes) == self.num_features_mulhot)
        self._embedding_size_list_mulhot = sizes
    elif isinstance(sizes, int):
      self._embedding_size_list_cat = [sizes] * self.num_features_cat
      self._embedding_size_list_mulhot = [sizes] * self.num_features_mulhot
    else:
      print('error: sizes need to be list or int')
      exit(0)
    return

  def set_target_prediction(self, features_cat_tr, full_values_tr,
    full_segids_tr, full_lengths_tr):
    # TODO: move these indices outside this class
    self.full_cat_tr = features_cat_tr
    self.full_values_tr = full_values_tr
    self.full_segids_tr = full_segids_tr
    self.full_lengths_tr = full_lengths_tr
    return

  # def get_item_last_index(self):
  #   return len(self.features_cat[0]) - 1

  def overview(self, out=None):
```

```
def overview(self, out=None):
  def p(val):
    if out:
      out(val)
    else:
      print(val)
  p('# of categorical attributes: {}'.format(self.num_features_cat))
  p('# of multi-hot   attributes: {}'.format(self.num_features_mulhot))
  p('====attributes values===')
  if self.num_features_cat > 0:
    p('\tinput categorical:')
    p('\t{}'.format(self.features_cat))
    if hasattr(self, 'full_cat_tr'):
      p('\toutput categorical:')
      p('\t{}'.format(self.full_cat_tr))
  if self.num_features_mulhot > 0:
    p('\tinput multi-hot:')
    p('\t values: {}'.format(self.features_mulhot))
    p('\t starts:{}'.format(self.mulhot_starts))
    p('\t length:{}'.format(self.mulhot_lengths))
    if hasattr(self, 'full_values_tr'):
      p('\toutput multi-hot:')
      p('\t values:{}'.format(self.full_values_tr))
      p('\t starts:{}'.format(self.full_segids_tr))
      p('\t length:{}'.format(self.full_lengths_tr))
  p('\n')

```

Embed_attribute.py

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
from six.moves import xrange  # pylint: disable=redefined-builtin
import tensorflow as tf
from tensorflow.python.framework import ops
from tensorflow.python.ops import variable_scope as vs
from tensorflow.python.ops import init_ops
from tensorflow.python.ops import embedding_ops
from tensorflow.python.ops import array_ops
from tensorflow.python.ops.embedding_ops import embedding_lookup as lookup
import itertools

from mulhot_index import *

class EmbeddingAttribute(object):
  def __init__(self, user_attributes, item_attributes, mb, n_sampled,
    input_steps=0, item_output=False,
```

```
item_ind2logit_ind=None, logit_ind2item_ind=None, indices_item=None,
devices=['/gpu:0']):
self.user_attributes = user_attributes
self.item_attributes = item_attributes
self.batch_size = mb
self.n_sampled = n_sampled
self.input_steps = input_steps
self.item_output = item_output # whether to use separate embedding for item output
self.num_item_features = (item_attributes.num_features_cat +
  item_attributes.num_features_mulhot)
self.reuse_item_tr = None

self.item_ind2logit_ind = item_ind2logit_ind
self.logit_ind2item_ind = logit_ind2item_ind
if logit_ind2item_ind is not None:
  self.logit_size = len(logit_ind2item_ind)
if indices_item is not None:
  self.indices_item = indices_item
else:
  self.indices_item = range(self.logit_size)
# self.logit_size_test = logit_size_test
self.mask = {}
self.zero_logits = {}
self.pos_indices = {}
self.l_true = {}
self.l_false = {}

self.devices = devices

self.att = {}
self._init_attributes(user_attributes, name='user', device=devices[0])
self._init_attributes(item_attributes, name='item', device=devices[0])
if self.item_output:
  self._init_attributes(item_attributes, name='item_output',
    device=devices[-1])

# user embeddings
self.user_embs_cat, self.user_embs_mulhot = self._embedded(user_attributes,
  prefix='user', device=devices[0])
#item embeddings
self.item_embs_cat, self.item_embs_mulhot = self._embedded(item_attributes,
  prefix='item', transpose=False, device=devices[0])
self.i_biases_cat, self.i_biases_mulhot = self._embedded_bias(
  item_attributes, 'item', device=devices[0])
if item_output:
  self.item_embs2_cat, self.item_embs2_mulhot = self._embedded(
```

```
    item_attributes, prefix='item_output', transpose=False, device=devices[-1])
  self.i_biases2_cat, self.i_biases2_mulhot = self._embedded_bias(
    item_attributes, 'item_output', device=devices[-1])

# input users
self.u_indices = {}
self.u_indices['input'] = self._placeholders('user', 'input', mb, device=devices[0])

self.i_indices = {}

# item -- positive/negative sample indices
print("construct postive/negative items/scores ")
self.i_indices['pos'] = self._placeholders('item', 'pos', mb, device=devices[0])
self.i_indices['neg'] = self._placeholders('item', 'neg', mb, device=devices[0])

# mini-batch item candidate pool
print("construct mini-batch item candicate pool")
if self.n_sampled is not None:
  self.i_indices['sampled_pass'] = self._placeholders('item', 'sampled',
    self.n_sampled, device=devices[-1])

# input items (for lstm etc)
print("construct input item")
for step in xrange(input_steps):
  name_ = 'input{}'.format(step)
  self.i_indices[name_] = self._placeholders('item', name_, mb, device=devices[0])

# item for prediction
''' full version'''
with tf.device(devices[-1]):
  ia = item_attributes
  print("construct full prediction layer")
  indices_cat, indices_mulhot, segids_mulhot, lengths_mulhot = [],[],[],[]
  for i in xrange(ia.num_features_cat):
    indices_cat.append(tf.constant(ia.full_cat_tr[i]))
  for i in xrange(ia.num_features_mulhot):
    indices_mulhot.append(tf.constant(ia.full_values_tr[i]))
    segids_mulhot.append(tf.constant(ia.full_segids_tr[i]))
    lengths_mulhot.append(tf.constant(ia.full_lengths_tr[i]))
  self.i_indices['full'] = (indices_cat, indices_mulhot, segids_mulhot,
    lengths_mulhot)

''' sampled version '''
print("sampled prediction layer")
if self.n_sampled is not None:
  prefix = 'item_output' if self.item_output else 'item'
```

```
    self.i_indices['sampled'] = self._var_indices(self.n_sampled,
      device=devices[-1])
    self.update_sampled = self._pass_sampled_items(prefix, device=devices[-1])
  return

def _var_indices(self, size, name='sampled', opt='item', device='/gpu:0'):
  cat_indices, mulhot_indices, mulhot_segids, mulhot_lengths = [],[], [], []
  att = self.item_attributes
  with tf.device(device):
    init_int32 = tf.constant(0)
    for i in xrange(att.num_features_cat):
      cat_indices.append(tf.get_variable(dtype = tf.int32,
        name = "var{}_{}_cat_ind_{}".format(opt, name, i), trainable=False,
        initializer=tf.zeros([size],dtype=tf.int32)))
    for i in xrange(att.num_features_mulhot):
      l1 = len(att.full_values_tr[i])
      mulhot_indices.append(tf.get_variable(dtype = tf.int32, trainable=False,
        initializer=tf.zeros([l1],dtype=tf.int32),
        name = "var{}_{}_mulhot_ind_{}".format(opt, name, i)))
      l2 = len(att.full_segids_tr[i])
      assert(l1==l2), 'length of indices/segids should be the same %d/%d'%(l1,l2)
      mulhot_segids.append(tf.get_variable(dtype = tf.int32, trainable=False,
        initializer=tf.zeros([l2],dtype=tf.int32),
        name = "var{}_{}_mulhot_seg_{}".format(opt, name, i)))
      mulhot_lengths.append(tf.get_variable(dtype =tf.float32, shape= [size, 1],
        name = "var{}_{}_mulhot_len_{}".format(opt, name, i), trainable=False))
    return (cat_indices, mulhot_indices, mulhot_segids, mulhot_lengths)

def _placeholders(self, opt, name, size, device='/gpu:0'):
  with tf.device(device):
    r = tf.placeholder(tf.int32, shape=[size], name = "{}_{}_ind".format(opt,
    name))
  return r

def get_prediction(self, latent, pool='full', device='/gpu:0', output_feat=1):
  '''
  output_feat: in prediction stage
    0: not using attributes
    1: using attributes, use mean to combine multi-hot features
    2: using attributes, use max  to combine multi-hot features
    3: same as 2, but softmax (instead of max)
  '''
  # compute inner product between item_hidden and {user_feature_embedding}
  # then lookup to compute logits
  with tf.device(device):
    out_layer = self.i_indices[pool]
```

```
indices_cat, indices_mulhot, segids_mulhot, lengths_mulhot = out_layer
innerps = []

n1 = 1 if output_feat == 0 else self.item_attributes.num_features_cat
n2 = 0 if output_feat == 0 else self.item_attributes.num_features_mulhot

for i in xrange(n1):
  item_emb_cat = self.item_embs2_cat[i] if self.item_output else self.item_embs_cat[i]
  i_biases_cat = self.i_biases2_cat[i] if self.item_output else self.i_biases_cat[i]
  u = latent[i] if isinstance(latent, list) else latent
  inds = indices_cat[i]
  innerp = tf.matmul(item_emb_cat, tf.transpose(u)) + i_biases_cat # Vf by mb
  innerps.append(lookup(innerp, inds)) # V by mb
offset = self.item_attributes.num_features_cat

for i in xrange(n2):
  item_embs_mulhot = self.item_embs2_mulhot[i] if self.item_output else
self.item_embs_mulhot[i]
  item_biases_mulhot = self.i_biases2_mulhot[i] if self.item_output else
self.i_biases_mulhot[i]
  u = latent[i+offset] if isinstance(latent, list) else latent
  lengs = lengths_mulhot[i]
  if pool == 'full':
    inds = indices_mulhot[i]
    segids = segids_mulhot[i]
    V = self.logit_size
  else:
    inds = tf.slice(indices_mulhot[i], [0], [self.sampled_mulhot_l[i]])
    segids = tf.slice(segids_mulhot[i], [0], [self.sampled_mulhot_l[i]])
    V = self.n_sampled
  innerp = tf.add(tf.matmul(item_embs_mulhot, tf.transpose(u)),
    item_biases_mulhot)

  if output_feat == 1:
    innerps.append(tf.div(tf.unsorted_segment_sum(lookup(innerp,
      inds), segids, V), lengs))
  elif output_feat == 2:
    innerps.append(tf.segment_max(lookup(innerp, inds), segids))
  elif output_feat == 3:
    score_max = tf.reduce_max(innerp)
    innerp = tf.subtract(innerp, score_max)
    innerps.append(score_max + tf.log(1 + tf.unsorted_segment_sum(tf.exp(
      lookup(innerp, inds)), segids, V)))
  else:
    print('Error: Attribute combination not implemented!')
    exit(1)
```

```
      logits = tf.transpose(tf.reduce_mean(innerps, 0))
    return logits

  def get_target_score(self, latent, inds, device='/gpu:0'):
    ''' TODO: max-pooling bug'''
    item_emb_cat = self.item_embs2_cat if self.item_output else self.item_embs_cat
    i_biases_cat = self.i_biases2_cat if self.item_output else self.i_biases_cat
    item_embs_mulhot = self.item_embs2_mulhot if self.item_output else
self.item_embs_mulhot
    item_biases_mulhot = self.i_biases2_mulhot if self.item_output else self.i_biases_mulhot

    cat_l, mulhot_l, i_bias = self._get_embedded(item_emb_cat, item_embs_mulhot,
      i_biases_cat, item_biases_mulhot, inds, self.batch_size,
      self.item_attributes, 'item', concatenation=False, device=device)
    with tf.device(device):
      target_item_emb = tf.reduce_mean(cat_l + mulhot_l, 0)
      return tf.reduce_sum(tf.multiply(latent, target_item_emb), 1) + i_bias

  def get_batch_user(self, keep_prob, concat=True, no_id=False, device='/gpu:0'):
    u_inds = self.u_indices['input']
    with tf.device(device):
      if concat:
        embedded_user, user_b = self._get_embedded(self.user_embs_cat,
          self.user_embs_mulhot, b_cat=None, b_mulhot=None, inds=u_inds,
          mb=self.batch_size, attributes=self.user_attributes, prefix='user',
          concatenation=concat, no_id=no_id, device=device)
      else:
        user_cat, user_mulhot, user_b = self._get_embedded(
          self.user_embs_cat, self.user_embs_mulhot, b_cat=None, b_mulhot=None,
          inds=u_inds, mb=self.batch_size, attributes=self.user_attributes,
          prefix='user', concatenation=concat, no_id=no_id, device=device)
        embedded_user =  tf.reduce_mean(user_cat + user_mulhot, 0)
      embedded_user = tf.nn.dropout(embedded_user, keep_prob)
    return embedded_user, user_b

  def get_batch_item(self, name, batch_size, concat=False, keep_prob=1.0,
    no_attribute = False, device='/gpu:0'):
    assert(name in self.i_indices)
    assert(keep_prob == 1.0), 'otherwise not implemented'
    i_inds = self.i_indices[name]
    if concat:
      return self._get_embedded(self.item_embs_cat, self.item_embs_mulhot,
        self.i_biases_cat, self.i_biases_mulhot, i_inds, batch_size,
        self.item_attributes, 'item', True,
        no_attribute=no_attribute, device=device)
```

```
  else:
    item_cat, item_mulhot, item_b = self._get_embedded(self.item_embs_cat,
      self.item_embs_mulhot, self.i_biases_cat, self.i_biases_mulhot, i_inds,
      batch_size, self.item_attributes, 'item', False,
      no_attribute=no_attribute, device=device)
    return item_cat + item_mulhot, item_b

def get_sampled_item(self, n_sampled, device='/gpu:0'):
  name = 'sampled'
  mapping = self.i_indices[name]
  with tf.device(device):
    item_cat, item_mulhot, item_b = self._get_embedded_sampled(
      self.item_embs_cat, self.item_embs_mulhot, self.i_biases_cat,
      self.i_biases_mulhot, mapping, n_sampled, self.item_attributes)
    return tf.reduce_mean(item_cat + item_mulhot, 0), item_b

def _embedded(self, attributes, prefix='', transpose=False, device='/gpu:0'):
  '''
  variables of full vocabulary for each type of features
  '''
  with tf.device(device):
    embs_cat, embs_mulhot = [], []
    for i in xrange(attributes.num_features_cat):
      d = attributes._embedding_size_list_cat[i]
      V = attributes._embedding_classes_list_cat[i]
      if not transpose:
        embedding = tf.get_variable(name=prefix + "embed_cat_{0}".format(i),
          shape=[V,d], dtype=tf.float32)
      else:
        embedding = tf.get_variable(name=prefix + "embed_cat_{0}".format(i),
          shape=[d,V], dtype=tf.float32)
      embs_cat.append(embedding)
    for i in xrange(attributes.num_features_mulhot):
      d = attributes._embedding_size_list_mulhot[i]
      V = attributes._embedding_classes_list_mulhot[i]
      if not transpose:
        embedding = tf.get_variable(name=prefix + "embed_mulhot_{0}".format(i),
          shape=[V,d], dtype=tf.float32)
      else:
        embedding = tf.get_variable(name=prefix + "embed_mulhot_{0}".format(i),
          shape=[d,V], dtype=tf.float32)
      embs_mulhot.append(embedding)
  return embs_cat, embs_mulhot

def _embedded_bias(self, attributes, prefix, device='/gpu:0'):
  with tf.device(device):
```

```
    biases_cat, biases_mulhot = [], []
    for i in range(attributes.num_features_cat):
      V = attributes._embedding_classes_list_cat[i]
      b = tf.get_variable(prefix + "_bias_cat_{0}".format(i), [V, 1],
        dtype = tf.float32)
      biases_cat.append(b)
    for i in range(attributes.num_features_mulhot):
      V = attributes._embedding_classes_list_mulhot[i]
      b = tf.get_variable(prefix + "_bias_mulhot_{0}".format(i), [V, 1],
        dtype = tf.float32)
      biases_mulhot.append(b)
  return biases_cat, biases_mulhot

def _init_attributes(self, att, name='user', device='/gpu:0'):
  features_cat, features_mulhot, mulhot_starts, mulhot_lengths=[],[],[],[]
  with tf.device(device):
    for i in range(att.num_features_cat):
      features_cat.append(tf.constant(att.features_cat[i], dtype=tf.int32))
    for i in range(att.num_features_mulhot):
      features_mulhot.append(tf.constant(att.features_mulhot[i], dtype=tf.int32))
      mulhot_starts.append(tf.constant(att.mulhot_starts[i], dtype=tf.int32))
      mulhot_lengths.append(tf.constant(att.mulhot_lengths[i], dtype=tf.int32))
    self.att[name] = (features_cat, features_mulhot, mulhot_starts,
      mulhot_lengths)

def _pass_sampled_items(self, prefix='item', device='/gpu:0'):
  self.sampled_mulhot_l = []
  res = []
  var_s = self.i_indices['sampled']
  att = self.item_attributes
  inds = self.i_indices['sampled_pass']
  with tf.device(device):
    for i in xrange(att.num_features_cat):
      vals = lookup(self.att[prefix][0][i], inds)
      res.append(tf.assign(var_s[0][i], vals))
    for i in xrange(att.num_features_mulhot):
      begin_ = lookup(self.att[prefix][2][i], inds)
      size_ = lookup(self.att[prefix][3][i], inds)
      b = tf.unstack(begin_)
      s = tf.unstack(size_)
      mulhot_indices = batch_slice2(self.att[prefix][1][i], b, s, self.n_sampled)
      mulhot_segids = batch_segids2(s, self.n_sampled)

      l0 = tf.reduce_sum(size_)
      indices = tf.range(l0)
      res.append(tf.scatter_update(var_s[1][i], indices, mulhot_indices))
```

```
        res.append(tf.scatter_update(var_s[2][i], indices, mulhot_segids))
        res.append(tf.assign(var_s[3][i], tf.reshape(tf.to_float(size_), [self.n_sampled, 1])))

        l = tf.get_variable(name='sampled_l_mulhot_{}'.format(i), dtype=tf.int32,
          initializer=tf.constant(0), trainable=False)
        self.sampled_mulhot_l.append(l)
        res.append(tf.assign(l, l0))
    return res

  def _get_embedded(self, embs_cat, embs_mulhot, b_cat, b_mulhot,
    inds, mb, attributes, prefix='', concatenation=True, no_id=False,
    no_attribute=False, device='/gpu:0'):
    cat_list, mulhot_list = [], []
    bias_cat_list, bias_mulhot_list = [], []
    with tf.device(device):
      if no_id and attributes.num_features_cat == 1:
        if b_cat is not None or b_mulhot is not None:
          print('error: not implemented')
          exit()
        bias = None
        dim = attributes._embedding_size_list_cat[0]
        cat_list = [tf.zeros([mb, dim],dtype=tf.float32)]
        if concatenation:
          return cat_list[0], bias
        else:
          return cat_list, [], bias

      n1 = 1 if no_attribute else attributes.num_features_cat
      n2 = 0 if no_attribute else attributes.num_features_mulhot

      for i in xrange(n1):
        if no_id and i == 0:
          continue
        cat_indices = lookup(self.att[prefix][0][i], inds)
        embedded = lookup(embs_cat[i], cat_indices,
          name='emb_lookup_item_{0}'.format(i))  # on cpu?
        cat_list.append(embedded)
        if b_cat is not None:
          b = lookup(b_cat[i], cat_indices,
            name = 'emb_lookup_item_b_{0}'.format(i))
          bias_cat_list.append(b)
      for i in xrange(n2):
        begin_ = lookup(self.att[prefix][2][i], inds)
        size_ = lookup(self.att[prefix][3][i], inds)
        # mulhot_indices, mulhot_segids = batch_slice_segids(
        #   self.att[prefix][1][i], begin_, size_, mb)
```

```
      # mulhot_indices = batch_slice(self.att[prefix][1][i], begin_,
      #   size_, mb)
      # mulhot_segids = batch_segids(size_, mb)

      b = tf.unstack(begin_)
      s = tf.unstack(size_)
      mulhot_indices = batch_slice2(self.att[prefix][1][i], b,
        s, mb)
      mulhot_segids = batch_segids2(s, mb)
      embedded_flat = lookup(embs_mulhot[i], mulhot_indices)
      embedded_sum = tf.unsorted_segment_sum(embedded_flat, mulhot_segids, mb)
      lengs = tf.reshape(tf.to_float(size_), [mb, 1])
      embedded = tf.div(embedded_sum, lengs)
      mulhot_list.append(embedded)
      if b_mulhot is not None:
        b_embedded_flat = lookup(b_mulhot[i], mulhot_indices)
        b_embedded_sum = tf.unsorted_segment_sum(b_embedded_flat, mulhot_segids,
          mb)
        b_embedded = tf.div(b_embedded_sum, lengs)
        bias_mulhot_list.append(b_embedded)

    if b_cat is None and b_mulhot is None:
      bias = None
    else:
      bias = tf.squeeze(tf.reduce_mean(bias_cat_list + bias_mulhot_list, 0))

    if concatenation:
      return concat_versions(1, cat_list + mulhot_list), bias
    else:
      return cat_list, mulhot_list, bias

def _get_embedded2(self, embs_cat, embs_mulhot, b_cat, b_mulhot,
  inds, mb, attributes, prefix='', concatenation=True, no_id=False,
  device='/gpu:0'):
  cat_list, mulhot_list = [], []
  bias_cat_list, bias_mulhot_list = [], []
  with tf.device(device):
    if no_id and attributes.num_features_cat == 1:
      if b_cat is not None or b_mulhot is not None:
        print('error: not implemented')
        exit()
      bias = None
      dim = attributes._embedding_size_list_cat[0]
      cat_list = [tf.zeros([mb, dim],dtype=tf.float32)]
      if concatenation:
```

```
    return cat_list[0], bias
  else:
    return cat_list, [], bias

for i in xrange(attributes.num_features_cat):
  if no_id and i == 0:
    continue
  cat_indices = lookup(self.att[prefix][0][i], inds)
  embedded = lookup(embs_cat[i], cat_indices,
    name='emb_lookup_item_{0}'.format(i))  # on cpu?
  cat_list.append(embedded)
  if b_cat is not None:
    b = lookup(b_cat[i], cat_indices,
      name = 'emb_lookup_item_b_{0}'.format(i))
    bias_cat_list.append(b)
for i in xrange(attributes.num_features_mulhot):
  begin_ = tf.unstack(lookup(self.att[prefix][2][i], inds))
  size_ = tf.unstack(lookup(self.att[prefix][3][i], inds))
  mulhot_i = []
  b_mulhot_i = []
  for j in xrange(mb):
    b = begin_[j]
    s = size_[j]
    m_inds = tf.slice(self.att[prefix][1][i], [b], [s])
    mulhot_i.append(tf.reduce_mean(lookup(embs_mulhot[i], m_inds), 0))
    # mulhot_i.append(tf.reduce_mean(lookup(embs_mulhot[i], m_inds), 0, True))
    if b_mulhot is not None:
      b_mulhot_i.append(tf.reduce_mean(lookup(b_mulhot[i], m_inds), 0))
      # b_mulhot_i.append(tf.reduce_mean(lookup(b_mulhot[i], m_inds), 0,
      #   True))
  # mulhot_list.append(concat_versions(0, mulhot_i))
  mulhot_list.append(tf.stack(mulhot_i))
  if b_mulhot is not None:
    # bias_mulhot_list.append(concat_versions(0, b_mulhot_i))
    bias_mulhot_list.append(tf.stack(b_mulhot_i))

if b_cat is None and b_mulhot is None:
  bias = None
else:
  bias = tf.squeeze(tf.reduce_mean(bias_cat_list + bias_mulhot_list, 0))

if concatenation:
  return concat_versions(1, cat_list + mulhot_list), bias
else:
  return cat_list, mulhot_list, bias
```

```
  def _get_embedded_sampled(self, embs_cat, embs_mulhot, b_cat, b_mulhot,
    mappings, n_sampled, attributes, device='/gpu:0'):
    cat_indices, mulhot_indices, mulhot_segids, mulhot_lengths = mappings
    cat_list, mulhot_list = [], []
    bias_cat_list, bias_mulhot_list = [], []
    with tf.device(device):
      for i in xrange(attributes.num_features_cat):
        embedded = lookup(embs_cat[i], cat_indices[i])
        cat_list.append(embedded)
        if b_cat is not None:
          b = lookup(b_cat[i], cat_indices[i])
          bias_cat_list.append(b)
      for i in xrange(attributes.num_features_mulhot):
        inds = tf.slice(mulhot_indices[i], [0], [self.sampled_mulhot_l[i]])
        segids = tf.slice(mulhot_segids[i], [0], [self.sampled_mulhot_l[i]])
        embedded_flat = lookup(embs_mulhot[i], inds)
        embedded_sum = tf.unsorted_segment_sum(embedded_flat, segids, n_sampled)
        embedded = tf.div(embedded_sum, mulhot_lengths[i])
        mulhot_list.append(embedded)
        if b_mulhot is not None:
          b_embedded_flat = lookup(b_mulhot[i], inds)
          b_embedded_sum = tf.unsorted_segment_sum(b_embedded_flat,
            segids, n_sampled)
          b_embedded = tf.div(b_embedded_sum, mulhot_lengths[i])
          bias_mulhot_list.append(b_embedded)
      if b_cat is None and b_mulhot is None:
        bias = None
      else:
        bias = tf.squeeze(tf.reduce_mean(bias_cat_list + bias_mulhot_list, 0))
      return cat_list, mulhot_list, bias

  def get_user_model_size(self, no_id=False, concat=True):
    if concat == True:
      cat_start = 1 if no_id else 0
      return
(sum(self.user_attributes._embedding_size_list_cat[cat_start:self.user_attributes.num_feature
s_cat]) +

sum(self.user_attributes._embedding_size_list_mulhot[0:self.user_attributes.num_features_m
ulhot]))
    else:
      return self.user_attributes._embedding_size_list_cat[0]

  def get_item_model_size(self, concat=True):
    if concat:
```

```
      return
(sum(self.item_attributes._embedding_size_list_cat[0:self.item_attributes.num_features_cat])
+

sum(self.item_attributes._embedding_size_list_mulhot[0:self.item_attributes.num_features_
mulhot]))
    else:
      return self.item_attributes._embedding_size_list_cat[0]

  def compute_loss(self, logits, item_target, loss='ce', true_rank=False,
   loss_func='log', exp_p=1.005, device='/gpu:0'):
    assert(loss in ['ce', 'mce', 'warp','warp_eval',  'rs', 'rs-sig','rs-sig2', 'mw', 'bbpr', 'bpr', 'bpr-
hinge'])
    with tf.device(device):
      if loss == 'ce':
        return tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits,
          labels=item_target)
      elif loss in ['rs', 'rs-sig', 'rs-sig2', 'bbpr']:
        return self._compute_rs_loss(logits, item_target, loss=loss,
          tr=true_rank, loss_func=loss_func, exp_p = exp_p)
      elif loss == 'warp':
        return self._compute_warp_loss(logits, item_target)
      elif loss == 'mw':
        return self._compute_mw_loss(logits, item_target)
      # elif loss =='bbpr':
      #   return self._compute_bbpr_loss(logits, item_target)
      elif loss == 'bpr':
        return tf.log(1 + tf.exp(logits))
      elif loss == 'bpr-hinge':
        return tf.maximum(1 + logits, 0)
      elif loss == 'warp_eval':
          return self._compute_warp_eval_loss(logits, item_target)
      else:
        print('Error: not implemented other loss!!')
        exit(1)

  def _compute_rs_loss(self, logits, item_target, loss='rs', loss_func='log',
    exp_p = 1.005, tr=False):
    assert(loss in ['rs', 'rs-sig', 'bbpr', 'rs-sig2'])
    if loss not in self.mask:
      self._prepare_loss_vars(loss)

    # compute error =  error(logits - target_logits)
    V = self.logit_size
    mb = self.batch_size
    flat_matrix = tf.reshape(logits, [-1])
```

```
  idx_flattened = self.idx_flattened0 + item_target
  target_logits = tf.gather(flat_matrix, idx_flattened)
  target_logits = tf.reshape(target_logits, [mb, 1])

  if loss in ['rs', 'rs-sig']: # margin
    errors = tf.subtract(logits, target_logits) + 1
    errors = tf.nn.relu(errors)
  elif loss in ['bbpr', 'rs-sig2']:
    errors = tf.sigmoid(tf.subtract(logits, target_logits))

  # masking other possitive instances
  mask2 = tf.reshape(self.mask[loss], [mb, V])
  errors_masked = tf.where(mask2, errors, self.zero_logits[loss])

  # rs-sig: take margin rank, go through sigmoid. 0-->1/2,  inf-> 1
  if loss in ['rs-sig']:
    errors_masked = tf.sigmoid(errors_masked)
    errors_masked = errors_masked * 2 - 1
  # compute loss
  if loss in ['rs', 'rs-sig', 'rs-sig2']:
    if loss_func == 'log':
      l = tf.log(1 + tf.reduce_sum(errors_masked, 1))
    elif loss_func == 'exp':
      l = 1 - tf.pow(exp_p, - tf.reduce_sum(errors_masked, 1))
    elif loss_func == 'poly':
      l = tf.pow(tf.reduce_sum(errors_masked, 1), exp_p)
    elif loss_func == 'poly2':
      l = tf.pow(1 + tf.reduce_sum(errors_masked, 1), exp_p)
    elif loss_func == 'linear':
      l = tf.reduce_sum(errors_masked, 1)
    elif loss_func == 'square':
      l = tf.square(tf.reduce_sum(errors_masked, 1))
  elif loss in ['bbpr']:
    l = tf.reduce_sum(errors_masked, 1)

  if tr:
    errors_nomargin = tf.nn.relu(tf.subtract(logits, target_logits))
    errors_nomargin_masked = tf.where(
      mask2, errors_nomargin, self.zero_logits[loss])
    true_rank = tf.count_nonzero(errors_nomargin_masked, 1)
    return [errors_masked, true_rank]

  return l

def _compute_warp_loss(self, logits, item_target):
  loss = 'warp'
```

```
  if loss not in self.mask:
    self._prepare_loss_vars(loss)
  V = self.logit_size
  mb = self.batch_size
  flat_matrix = tf.reshape(logits, [-1])
  idx_flattened = self.idx_flattened0 + item_target
  logits_ = tf.gather(flat_matrix, idx_flattened)
  logits_ = tf.reshape(logits_, [mb, 1])
  logits2 = tf.subtract(logits, logits_) + 1
  mask2 = tf.reshape(self.mask[loss], [mb, V])
  target = tf.where(mask2, logits2, self.zero_logits[loss])
  return tf.log(1 + tf.reduce_sum(tf.nn.relu(target), 1))

def _compute_warp_eval_loss(self, logits, item_target):
  loss = 'warp_eval'
  if loss not in self.mask:
    self._prepare_loss_vars(loss)
  V = self.logit_size
  mb = self.batch_size
  flat_matrix = tf.reshape(logits, [-1])
  idx_flattened = self.idx_flattened0 + item_target
  logits_ = tf.gather(flat_matrix, idx_flattened)
  logits_ = tf.reshape(logits_, [mb, 1])
  logits2 = tf.subtract(logits, logits_) + 1
  mask2 = tf.reshape(self.mask[loss], [mb, V])
  target = tf.where(mask2, logits2, self.zero_logits[loss])
  margin_rank =  tf.reduce_sum(tf.nn.relu(target), 1)

  logits3 = tf.nn.relu(tf.subtract(logits, logits_))
  target2 = tf.where(mask2, logits3, self.zero_logits[loss])
  true_rank = tf.count_nonzero(target2, 1)

  return [margin_rank, true_rank]

def _compute_mw_loss(self, logits, item_target):
  if 'mw' not in self.mask:
    self._prepare_loss_vars('mw')
  V = self.n_sampled
  mb = self.batch_size
  logits2 = tf.subtract(logits, tf.reshape(item_target, [mb, 1])) + 1
  mask2 = tf.reshape(self.mask['mw'], [mb, V])
  target = tf.where(mask2, logits2, self.zero_logits['mw'])
  return tf.log(1 + tf.reduce_sum(tf.nn.relu(target), 1)) # scale or not??

def _prepare_loss_vars(self, loss= 'warp'):
  V = self.n_sampled if loss == 'mw' else self.logit_size
```

```
  mb = self.batch_size
  self.idx_flattened0 = tf.range(0, mb) * V
  self.mask[loss] = tf.Variable([True] * V * mb, dtype=tf.bool,
    trainable=False)
  self.zero_logits[loss] = tf.constant([[0.0] * V] * mb)
  self.pos_indices[loss] = tf.placeholder(tf.int32, shape = [None])
  self.l_true[loss] = tf.placeholder(tf.bool, shape = [None], name='l_true')
  self.l_false[loss] = tf.placeholder(tf.bool, shape = [None], name='l_false')

def get_warp_mask(self, device='/gpu:0'):
  self.set_mask, self.reset_mask = {}, {}
  with tf.device(device):
    for loss in ['mw', 'warp','warp_eval', 'rs', 'rs-sig', 'rs-sig2', 'bbpr']:
      if loss not in self.mask:
        continue
      self.set_mask[loss] = tf.scatter_update(self.mask[loss],
        self.pos_indices[loss], self.l_false[loss])
      self.reset_mask[loss] = tf.scatter_update(self.mask[loss],
        self.pos_indices[loss], self.l_true[loss])
    return self.set_mask, self.reset_mask

def prepare_warp(self, pos_item_set, pos_item_set_eval):
  self.pos_item_set = pos_item_set
  self.pos_item_set_eval = pos_item_set_eval
  return

def target_mapping(self, item_target):
  m = self.item_ind2logit_ind
  target = []
  for items in item_target:
    target.append([m[v] for v in items])
  return target

def _add_input(self, input_feed, opt, input_, name_):
  if opt == 'user':
    att = self.user_attributes
    mappings = self.u_indices[name_]
  elif opt == 'item':
    att = self.item_attributes
    mappings = self.i_indices[name_]
  else:
    exit(-1)
  input_feed[mappings.name] = input_

def add_input(self, input_feed, user_input, item_input,
      neg_item_input=None, item_sampled = None, item_sampled_id2idx = None,
```

```
        forward_only=False, recommend=False, loss=None):
    # users
    if self.user_attributes is not None:
      self._add_input(input_feed, 'user', user_input, 'input')
    # pos
    # if self.item_attributes is not None and recommend is False and self.input_steps == 0:
    #   self._add_input(input_feed, 'item', item_input, 'pos')
    #   self._add_input(input_feed, 'item', neg_item_input, 'neg')

    # input item: for lstm, skipgram,
    if self.item_attributes is not None and self.input_steps > 0:
      for step in range(len(item_input)):
        self._add_input(input_feed, 'item', item_input[step],
          'input{}'.format(step))

    # sampled item: when sampled-loss is used
    input_feed_sampled = {}
    update_sampled = []
    if self.item_attributes is not None and recommend is False and item_sampled is not None
and loss in ['mw', 'mce']:
      self._add_input(input_feed_sampled, 'item', item_sampled, 'sampled_pass')
      update_sampled = self.update_sampled

    # for warp loss.
    input_feed_warp = {}
    if loss in ['warp', 'warp_eval', 'mw', 'rs', 'rs-sig','rs-sig2', 'bbpr'] and recommend is False:
      V = self.n_sampled if loss == 'mw' else self.logit_size
      mask_indices, c = [], 0
      s_2idx = self.item_ind2logit_ind if loss in ['warp', 'warp_eval', 'rs', 'rs-sig', 'rs-sig2', 'bbpr']
else item_sampled_id2idx
      item_set = self.pos_item_set_eval if forward_only else self.pos_item_set

      if loss in ['warp', 'warp_eval', 'bbpr', 'rs', 'rs-sig', 'rs-sig2']:
        for u in user_input:
          offset = c * V
          if u in item_set:
            mask_indices.extend([s_2idx[v] + offset for v in item_set[u]])
          c += 1
      else:
        for u in user_input:
          offset = c * V
          if u in item_set:
            mask_indices.extend([s_2idx[v] + offset for v in item_set[u]
              if v in s_2idx])
          c += 1
      L = len(mask_indices)
```

```
      input_feed_warp[self.pos_indices[loss].name] = mask_indices
      input_feed_warp[self.l_false[loss].name] = [False] * L
      input_feed_warp[self.l_true[loss].name] = [True] * L

    return update_sampled, input_feed_sampled, input_feed_warp
```

mulhot_index.py

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import tensorflow as tf

def concat_versions(axis, value):
  if tf.__version__.startswith('0'):
    return tf.concat(axis, value)
  else:
    return tf.concat(value, axis)

def batch_slice(target, begin, size, l):
  b = tf.unstack(begin)
  s = tf.unstack(size)
  res = []
  for i in range(l):
    res.append(tf.slice(target, [b[i]], [s[i]]))
  return concat_versions(0, res)

def batch_segids(size, l):
  s = tf.unstack(size)
  res = []
  for i in range(l):
    ok = tf.tile([i], [s[i]])
    res.append(ok)
  return concat_versions(0, res)

def batch_slice_segids(target, begin, size, l):
  b = tf.unstack(begin)
  s = tf.unstack(size)
  res = []
  res2 = []
  for i in range(l):
    res.append(tf.slice(target, [b[i]], [s[i]]))
    res2.append(tf.tile([i], [s[i]]))
  return concat_versions(0, res), concat_versions(0, res2)

def batch_slice20(target, b, s, l):
  res1, res2 = [], []
```

```
  h = int(l/2)
  assert(l/2 == h)
  for i in range(h):
    res1.append(tf.slice(target, [b[i]], [s[i]]))
    res2.append(tf.slice(target, [b[i+h]], [s[i+h]]))
  return concat_versions(0, res1+res2)

def batch_slice2(target, b, s, l):
  res = []
  for i in range(l):
    res.append(tf.slice(target, [b[i]], [s[i]]))
  return concat_versions(0, res)

def batch_segids20(s, l):
  res1, res2 = [], []
  h = int(l/2)
  for i in range(h):
    res1.append(tf.tile([i], [s[i]]))
    res2.append(tf.tile([i+h], [s[i+h]]))
  return concat_versions(0, res1 + res2)

def batch_segids2(s, l):
  res = []
  for i in range(l):
    ok = tf.tile([i], [s[i]])
    res.append(ok)
  return concat_versions(0, res)
```

comb_attribute.py

```
from preprocess import create_dictionary, create_dictionary_mix,
tokenize_attribute_map, filter_cat, filter_mulhot, pickle_save
import numpy as np
import attribute


class Comb_Attributes(object):
  def __init__(self):
    return

  def get_attributes(self, users, items, data_tr, user_features, item_features):
    # create_dictionary
    user_feature_names, user_feature_types = user_features
    item_feature_names, item_feature_types = item_features
```

```
    u_inds = [p[0] for p in data_tr]
    self.create_dictionary(self.data_dir, u_inds, users, user_feature_types,
      user_feature_names, self.max_vocabulary_size, self.logits_size_tr,
      prefix='user', threshold=self.threshold)

    # create user feature map
    (num_features_cat, features_cat, num_features_mulhot, features_mulhot,
      mulhot_max_leng, mulhot_starts, mulhot_lengs, v_sizes_cat,
      v_sizes_mulhot) = tokenize_attribute_map(self.data_dir, users,
      user_feature_types, self.max_vocabulary_size, self.logits_size_tr,
      prefix='user')

    u_attributes = attribute.Attributes(num_features_cat, features_cat,
      num_features_mulhot, features_mulhot, mulhot_max_leng, mulhot_starts,
      mulhot_lengs, v_sizes_cat, v_sizes_mulhot)

    # create_dictionary
    i_inds_tr = [p[1] for p in data_tr]
    self.create_dictionary(self.data_dir, i_inds_tr, items, item_feature_types,
      item_feature_names, self.max_vocabulary_size, self.logits_size_tr,
      prefix='item', threshold=self.threshold)

    # create item feature map
    items_cp = np.copy(items)
    (num_features_cat2, features_cat2, num_features_mulhot2,
features_mulhot2,
      mulhot_max_leng2, mulhot_starts2, mulhot_lengs2, v_sizes_cat2,
      v_sizes_mulhot2) = tokenize_attribute_map(self.data_dir,
      items_cp, item_feature_types, self.max_vocabulary_size, self.logits_size_tr,
      prefix='item')

    '''
    create an (item-index <--> classification output) mapping
    there are more than one valid mapping as long as 1 to 1
    '''
    item2fea0 = features_cat2[0] if len(features_cat2) > 0 else None
    item_ind2logit_ind, logit_ind2item_ind = self.index_mapping(item2fea0,
      i_inds_tr, len(items))
```

```
    i_attributes = attribute.Attributes(num_features_cat2, features_cat2,
      num_features_mulhot2, features_mulhot2, mulhot_max_leng2,
mulhot_starts2,
      mulhot_lengs2, v_sizes_cat2, v_sizes_mulhot2)

    # set target prediction indices
    features_cat2_tr = filter_cat(num_features_cat2, features_cat2,
      logit_ind2item_ind)

    (full_values, full_values_tr, full_segids, full_lengths, full_segids_tr,
      full_lengths_tr) = filter_mulhot(self.data_dir, items,
      item_feature_types, self.max_vocabulary_size, logit_ind2item_ind,
      prefix='item')

    i_attributes.set_target_prediction(features_cat2_tr, full_values_tr,
      full_segids_tr, full_lengths_tr)

    return u_attributes, i_attributes, item_ind2logit_ind, logit_ind2item_ind


class MIX(Comb_Attributes):

  def __init__(self, data_dir, max_vocabulary_size=500000,
logits_size_tr=50000,
    threshold=2):
    self.data_dir = data_dir
    self.max_vocabulary_size = max_vocabulary_size
    self.logits_size_tr = logits_size_tr
    self.threshold = threshold
    self.create_dictionary = create_dictionary_mix
    return

  def index_mapping(self, item2fea0, i_inds, M=None):
    item_ind2logit_ind = {}
    logit_ind2item_ind = {}

    item_ind_count = {}
    for i_ind in i_inds:
      item_ind_count[i_ind] = item_ind_count[i_ind] + 1 if i_ind in
item_ind_count else 1
```

```
    ind_list = sorted(item_ind_count, key=item_ind_count.get, reverse=True)
    assert(self.logits_size_tr <= len(ind_list)), 'Item_vocab_size should be
smaller than # of appeared items'
    ind_list = ind_list[:self.logits_size_tr]

    for index, elem in enumerate(ind_list):
      item_ind2logit_ind[elem] = index
      logit_ind2item_ind[index] = elem

    return item_ind2logit_ind, logit_ind2item_ind

  def mix_attr(self, users, items, user_features, item_features):
    user_feature_names, user_feature_types = user_features
    item_feature_names, item_feature_types = item_features
    user_feature_names[0] = 'uid'

    # user
    n = len(users)
    users2  = np.zeros((n, 1), dtype=object)
    for i in range(n):
      v = []
      user = users[i, :]
      for j in range(len(user_feature_types)):
        t = user_feature_types[j]
        n = user_feature_names[j]
        if t == 0:
          v.append(n + str(user[j]))
        elif t == 1:
          v.extend([n + s for s in user[j].split(',')])
        else:
          continue
      users2[i, 0] = ','.join(v)

    # item
    n = len(items)
    items2  = np.zeros((n, 1), dtype=object)
    for i in range(n):
      v = []
      item = items[i, :]
      for j in range(len(item_feature_types)):
```

```
        t = item_feature_types[j]
        n = item_feature_names[j]
        if t == 0:
          v.append(n + str(item[j]))
        elif t == 1:
          v.extend([n + s for s in item[j].split(',')])
        else:
          continue
      items2[i, 0] = ','.join(v)

    # modify attribute names and types
    if len(user_feature_types) == 1 and user_feature_types[0] == 0:
      user_features = ([['mix'], [0]])
    else:
      user_features = ([['mix'], [1]])
    if len(item_feature_types) == 1 and item_feature_types[0] == 0:
      item_features = ([['mix'], [0]])
    else:
      item_features = ([['mix'], [1]])
    return users2, items2, user_features, item_features


class HET(Comb_Attributes):

  def __init__(self, data_dir, max_vocabulary_size=50000,
logits_size_tr=50000,
    threshold=2):
    self.data_dir = data_dir
    self.max_vocabulary_size = max_vocabulary_size
    self.logits_size_tr = logits_size_tr
    self.threshold = threshold
    self.create_dictionary = create_dictionary
    return

  def index_mapping(self, item2fea0, i_inds, M):
    item_ind2logit_ind = {}
    logit_ind2item_ind = {}
    ind = 0
    for i in range(M):
      fea0 = item2fea0[i]
```

```
      if fea0 != 0:
        item_ind2logit_ind[i] = ind
        ind += 1
    assert(ind == self.logits_size_tr), 'Item_vocab_size %d too large! need to be
no greater than %d\nFix: --item_vocab_size [smaller item_vocab_size]\n' %
(self.logits_size_tr, ind)

    logit_ind2item_ind = {}
    for k, v in item_ind2logit_ind.items():
      logit_ind2item_ind[v] = k
    return item_ind2logit_ind, logit_ind2item_ind
```

Preprocess.py

```
import numpy as np
from os import listdir, mkdir, path, rename
from os.path import isfile, join
from tensorflow.python.platform import gfile

def pickle_save(m, filename):
  import cPickle as pickle
  pickle.dump(m, open(filename, 'wb'),
protocol=pickle.HIGHEST_PROTOCOL)

def initialize_vocabulary(vocabulary_path):
  Args:
    vocabulary_path: path to the file containing the vocabulary.

  Raises:
    ValueError: if the provided vocabulary_path does not exist.
  """
  if gfile.Exists(vocabulary_path):
    rev_vocab = []
    with gfile.GFile(vocabulary_path, mode="rb") as f:
      rev_vocab.extend(f.readlines())
    rev_vocab = [line.strip() for line in rev_vocab]
    vocab = dict([(x, y) for (y, x) in enumerate(rev_vocab)])
    return vocab, rev_vocab
  else:
    raise ValueError("Vocabulary file %s not found.", vocabulary_path)
```

```
def create_dictionary(data_dir, inds, features, feature_types, feature_names,
  max_vocabulary_size=50000, logits_size_tr = 50000, threshold = 2,
  prefix='user'):
  filename = 'vocab0_%d' % max_vocabulary_size
  if isfile(join(data_dir, filename)):
    print("vocabulary exists!")
    return
  vocab_counts = {}
  num_uf = len(feature_names)
  assert(len(feature_types) == num_uf), 'length of feature_types should be the
same length of feature_names {} vs {}'.format(len(feature_types), num_uf)
  for ind in range(num_uf):
    name = feature_names[ind]
    vocab_counts[name] = {}

  for u in inds: # u index
    uf = features[u, :]
    for ii in range(num_uf):
      name = feature_names[ii]
      if feature_types[ii] == 0:
        vocab_counts[name][uf[ii]] = vocab_counts[name][uf[ii]] + 1 if uf[ii] in
vocab_counts[name] else 1
      elif feature_types[ii] == 1:
        if not isinstance(uf[ii], list):
          if not isinstance(uf[ii], str):
            uf[ii] = str(uf[ii])
          uf[ii] = uf[ii].split(',')
        for t in uf[ii]:
          vocab_counts[name][t] = vocab_counts[name][t] + 1 if t in
vocab_counts[name] else 1

  minimum_occurance = []
  for i in range(num_uf):
    name = feature_names[i]
    if feature_types[i] > 1:
      continue
    vocab_list = _START_VOCAB + sorted(vocab_counts[name],
      key=vocab_counts[name].get, reverse=True)
    if prefix == 'item' and i == 0:
```

```
      max_size = logits_size_tr + len(_START_VOCAB)
    elif prefix == 'user' and i == 0:
      max_size = len(features)
      max_size = max_vocabulary_size # looks still better to filter first
    else:
      max_size = max_vocabulary_size

    # max_size += len(_START_VOCAB)

    # if len(vocab_list) > max_size:
    #   vocab_list= vocab_list[:max_size]
    with gfile.GFile(join(data_dir, ("%s_vocab%d_%d"% (prefix, i,
      max_size))), mode="wb") as vocab_file:

      if prefix == 'user' and i == 0:
        vocab_list2 = [v for v in vocab_list if v in _START_VOCAB or
         vocab_counts[name][v] >= threshold]
      else:
        vocab_list2 = [v for v in vocab_list if v in _START_VOCAB or
         vocab_counts[name][v] >= threshold]
      if len(vocab_list2) > max_size:
        print("vocabulary {}_{} longer than max_vocabulary_size {}. Truncate the
tail".format(prefix, len(vocab_list2), max_size))
        vocab_list2= vocab_list2[:max_size]
      for w in vocab_list2:
        vocab_file.write(str(w) + b"\n")
      minimum_occurance.append(vocab_counts[name][vocab_list2[-1]])
  with gfile.GFile(join(data_dir, "%s_minimum_occurance_%d" %(prefix,
    max_size)), mode="wb") as sum_file:
    sum_file.write('\n'.join([str(v) for v in minimum_occurance]))

  return

def create_dictionary_mix(data_dir, inds, features, feature_types,
  feature_names, max_vocabulary_size=50000, logits_size_tr = 50000,
  threshold = 2, prefix='user'):
  filename = 'vocab0_%d' % max_vocabulary_size
  if isfile(join(data_dir, filename)):
    print("vocabulary exists!")
    return
```

```
  vocab_counts = {}
  num_uf = len(feature_names)
  assert(len(feature_types) == num_uf), 'length of feature_types should be the
same length of feature_names {} vs {}'.format(len(feature_types), num_uf)

  vocab_uid = {}
  vocab = {}
  for u in inds: # u index
    uf = features[u, 0]

    if not isinstance(uf, list):
      uf = uf.split(',')
    for t in uf:
      if t.startswith('uid'):
        vocab_uid[t] = vocab_uid[t] + 1 if t in vocab_uid else 1
      else:
        vocab[t] = vocab[t] + 1 if t in vocab else 1

  minimum_occurance = []

  vocab_list = _START_VOCAB + vocab_uid.keys() + sorted(vocab,
    key=vocab.get, reverse=True)

  max_size = max_vocabulary_size

  with gfile.GFile(join(data_dir, ("%s_vocab%d_%d"% (prefix, 0,
    max_size))), mode="wb") as vocab_file:

    vocab_list2 = [v for v in vocab_list if v in _START_VOCAB or (v in vocab
and
      vocab[v] >= threshold) or (v in vocab_uid and vocab_uid[v] >= threshold)]
    if len(vocab_list2) > max_size:
      print("vocabulary {}_{} longer than max_vocabulary_size {}. Truncate the
tail".format(prefix, len(vocab_list2), max_size))
      vocab_list2 = vocab_list2[:max_size]

    for w in vocab_list2:
      vocab_file.write(str(w) + b"\n")
    min_occurance = vocab[vocab_list2[-1]] if vocab_list2[-1] in vocab else
vocab_uid[vocab_list2[-1]]
```

```
    minimum_occurance.append(min_occurance)
  with gfile.GFile(join(data_dir, "%s_minimum_occurance_%d" %(prefix,
    max_size)), mode="wb") as sum_file:
    sum_file.write('\n'.join([str(v) for v in minimum_occurance]))

  return

def tokenize_attribute_map(data_dir, features, feature_types,
max_vocabulary_size,
  logits_size_tr=50000, prefix='user'):
  """
  read feature maps and tokenize with loaded vocabulary
  output required format for Attributes
  """
  features_cat, features_mulhot = [], []
  v_sizes_cat, v_sizes_mulhot = [], []
  mulhot_max_leng, mulhot_starts, mulhot_lengs = [], [], []
  # logit_ind2item_ind = {}
  for i in range(len(feature_types)):
    ut = feature_types[i]
    if feature_types[i] > 1:
      continue

    path = "%s_vocab%d_" %(prefix, i)
    vocabulary_paths = [f for f in listdir(data_dir) if f.startswith(path)]
    assert(len(vocabulary_paths) == 1)
    vocabulary_path = join(data_dir, vocabulary_paths[0])

    vocab, _ = initialize_vocabulary(vocabulary_path)

    N = len(features)
    users2 = np.copy(features)
    uf = features[:, i]
    if ut == 0:
      v_sizes_cat.append(len(vocab))
      for n in range(N):
        uf[n] = vocab.get(str(uf[n]), UNK_ID)
      uf = np.append(uf, START_ID)
      features_cat.append(uf)
    else:
```

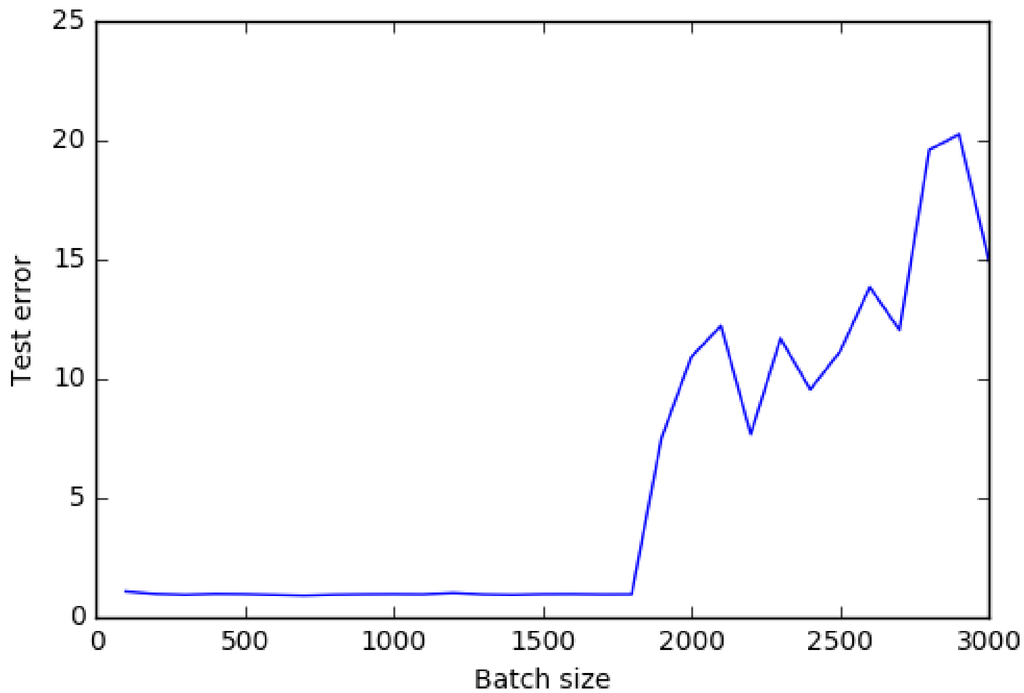
```
      mtl = 0
      idx = 0
      starts, lengs, vals = [idx], [], []
      v_sizes_mulhot.append(len(vocab))
      for n in range(N):
        elem = uf[n]
        if not isinstance(elem, list):
          if not isinstance(elem, str):
            elem = str(elem)
          elem = elem.split(',')
        val = [vocab.get(str(v), UNK_ID) for v in elem]
        val_ = [v for v in val if v != UNK_ID]
        if len(val_) == 0:
          val_ = [UNK_ID]

        vals.extend(val_)
        l_mulhot = len(val_)
        mtl = max(mtl, l_mulhot)
        idx += l_mulhot
        starts.append(idx)
        lengs.append(l_mulhot)

      vals.append(START_ID)
      idx += 1
      starts.append(idx)
      lengs.append(1)

      mulhot_max_leng.append(mtl)
      mulhot_starts.append(np.array(starts))
      mulhot_lengs.append(np.array(lengs))
      features_mulhot.append(np.array(vals))

  num_features_cat = sum(v == 0 for v in feature_types)
  num_features_mulhot= sum(v == 1 for v in feature_types)
  assert(num_features_cat + num_features_mulhot <= len(feature_types))
  return (num_features_cat, features_cat, num_features_mulhot,
features_mulhot,
    mulhot_max_leng, mulhot_starts, mulhot_lengs, v_sizes_cat,
    v_sizes_mulhot)
```

```
def filter_cat(num_features_cat, features_cat, logit_ind2item_ind):
  '''
  create mapping from logits index [0, logits_size) to features
  '''
  features_cat_tr = []
  size = len(logit_ind2item_ind)
  for i in xrange(num_features_cat):
    feat_cat = features_cat[i]
    feat_cat_tr = []
    for j in xrange(size):
      item_index = logit_ind2item_ind[j]
      feat_cat_tr.append(feat_cat[item_index])
    features_cat_tr.append(np.array(feat_cat_tr))

  return features_cat_tr


def filter_mulhot(data_dir, items, feature_types, max_vocabulary_size,
  logit_ind2item_ind, prefix='item'):
  full_values,  full_values_tr= [], []
  full_segids, full_lengths = [], []
  full_segids_tr, full_lengths_tr = [], []

  L = len(logit_ind2item_ind)
  N = len(items)
  for i in range(len(feature_types)):
    full_index, full_index_tr = [], []
    lengs, lengs_tr = [], []
    ut = feature_types[i]
    if feature_types[i] == 1:

      path = "%s_vocab%d_" %(prefix, i)
      vocabulary_paths = [f for f in listdir(data_dir) if f.startswith(path)]
      assert(len(vocabulary_paths)==1), 'more than one dictionaries found! delete
unnecessary ones to fix this.'
      vocabulary_path = join(data_dir, vocabulary_paths[0])

      vocab, _ = initialize_vocabulary(vocabulary_path)

      uf = items[:, i]
```

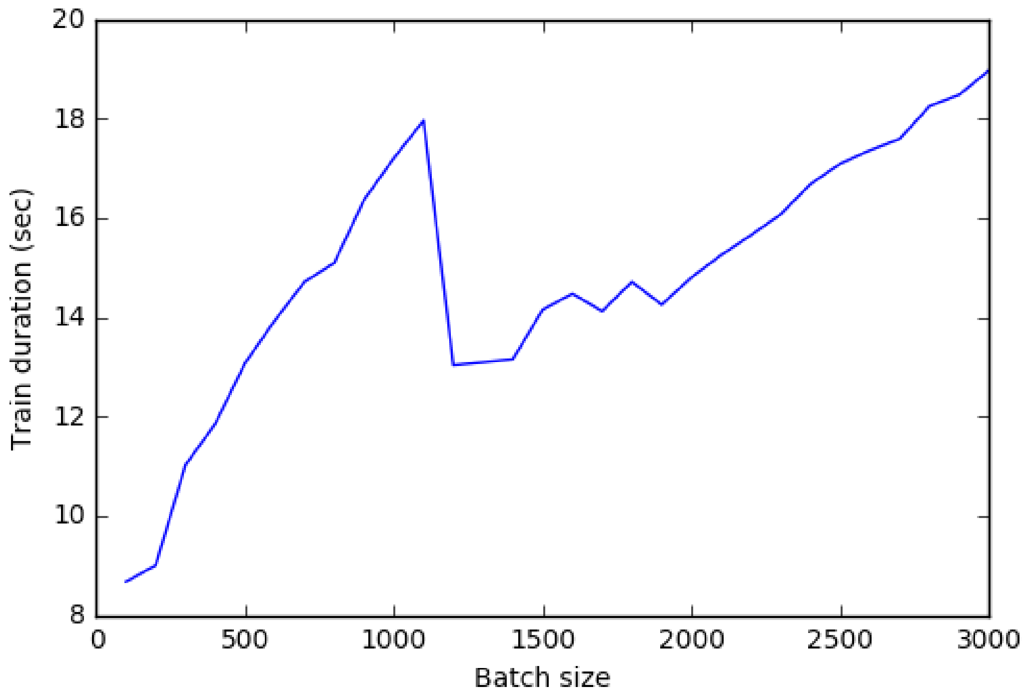
```
mtl, idx, vals = 0, 0, []
segids = []

for n in xrange(N):
  elem = uf[n]
  if not isinstance(elem, list):
    if not isinstance(elem, str):
      elem = str(elem)
    elem = elem.split(',')

  val = [vocab.get(v, UNK_ID) for v in elem]
  val_ = [v for v in val if v != UNK_ID]
  if len(val_) == 0:
    val_ = [UNK_ID]
  vals.extend(val_)
  l_mulhot = len(val_)
  segids.extend([n] * l_mulhot)
  lengs.append([l_mulhot * 1.0])

full_values.append(vals)
full_segids.append(segids)
full_lengths.append(lengs)

idx2, vals2 = 0, []
segids_tr = []
for n in xrange(L):
  i_ind = logit_ind2item_ind[n]
  elem = uf[i_ind]
  if not isinstance(elem, list):
    if not isinstance(elem, str):
      elem = str(elem)
    elem = elem.split(',')

  val = [vocab.get(v, UNK_ID) for v in elem]
  val_ = [v for v in val if v != UNK_ID]
  if len(val_) == 0:
    val_ = [UNK_ID]
  vals2.extend(val_)
  l_mulhot = len(val_)
  lengs_tr.append([l_mulhot * 1.0])
```

```
    segids_tr.extend([n] * l_mulhot)

   full_values_tr.append(vals2)
   full_segids_tr.append(segids_tr)
   full_lengths_tr.append(lengs_tr)

return (full_values, full_values_tr, full_segids, full_lengths,
  full_segids_tr, full_lengths_tr)
```

Output



Error when trained model with dataset containing negative dataset
Note: Batch size is 0k-3000k



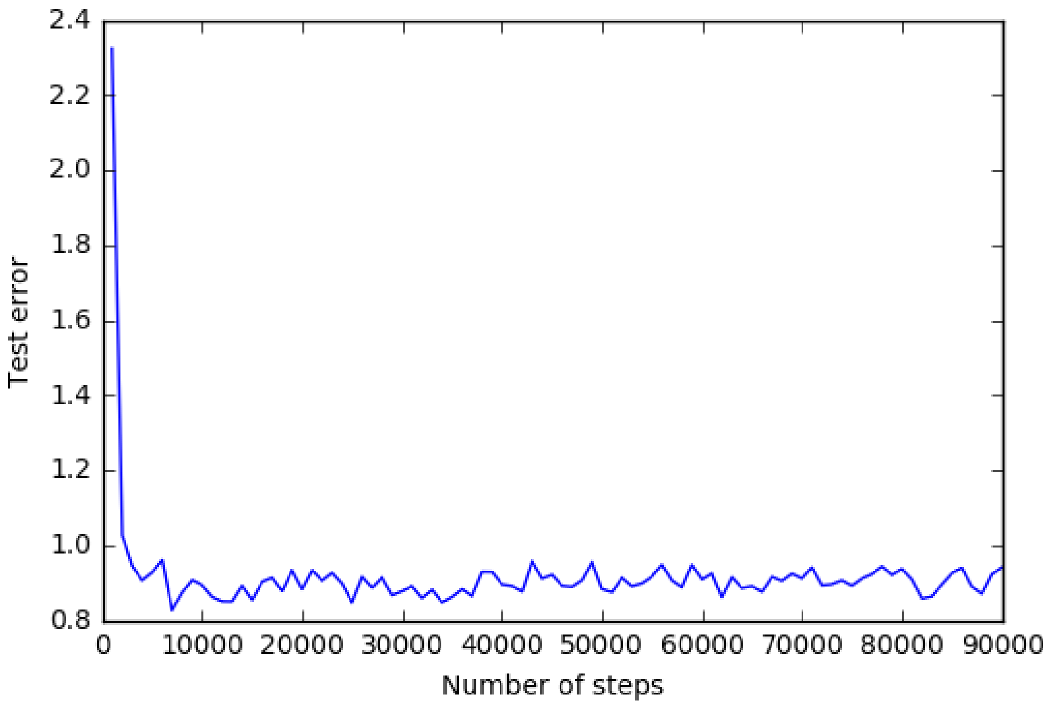
Plotted Duration of training

```
In [8]: all_batch_sizes = list(all_batch_sizes)
        best_result = min(list(zip(results,all_batch_sizes,times)))
        result_string = """In an experiment with batch sizes from {0} to {1}
        the best size for the mini batch is {2} with error {3}.
        Using this size the training will take {4} seconds""".format(all_batch_sizes[0],
                                                                      all_batch_sizes[-1:][0],
                                                                      best_result[1],
                                                                      best_result[0],
                                                                      best_result[2])
        print(result_string)

        In an experiment with batch sizes from 100 to 3000
        the best size for the mini batch is 700 with error 0.8733594417572021.
        Using this size the training will take 14.71 seconds

In [11]: print(np.mean(results),np.std(results))

         5.59026 6.21118

In [13]: print(np.mean(times),np.std(times))

         14.8673333333 2.54731745611
```

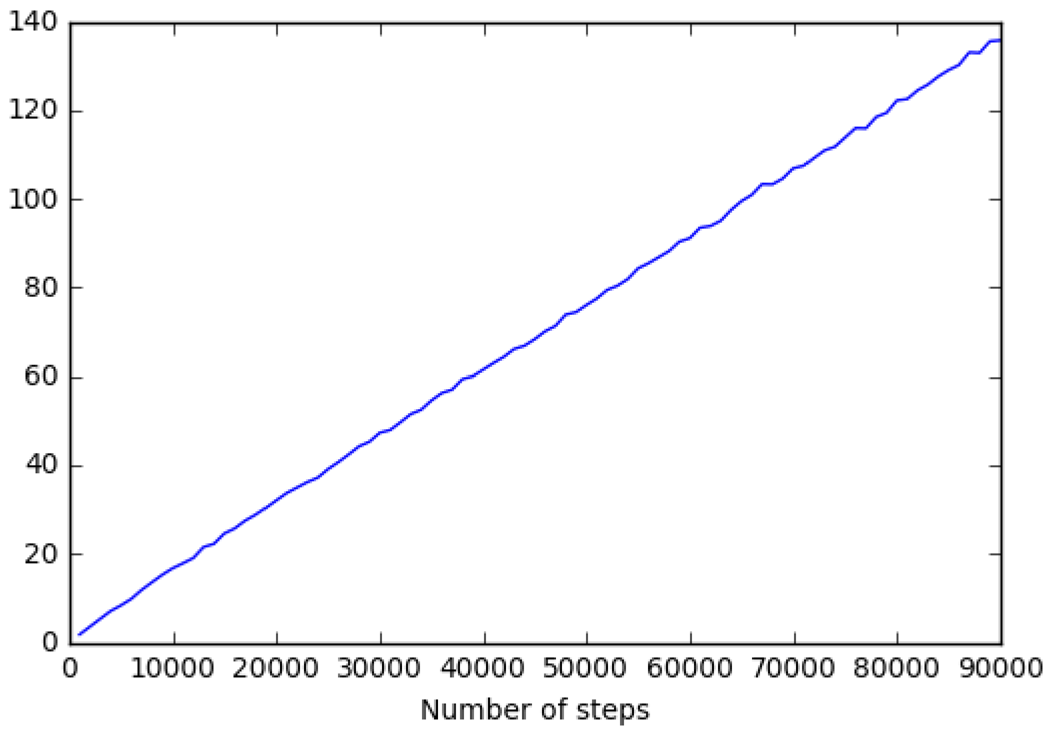
Intelligent batch dispatcher for fragment based training

```
In [3]: all_constants = list(all_constants)
        aggregate = list(zip(results,all_constants,times))
        best_result = min(aggregate)
        result_string = """In an experiment with 300 random constants
        the best momentum factor is {0} with error {1}.
        Using this constant the training will take {2} seconds""".format(
                                                                best_result[1],
                                                                best_result[0],
                                                                best_result[2])
        print(result_string)

        In an experiment with 300 random constants
        the best momentum factor is 0.9264120820573123 with error 0.8353284001350403.
        Using this constant the training will take 14.41 seconds

In [4]: print(np.mean(results),np.std(results))

        0.955692 0.15133

In [5]: print(np.mean(times),np.std(times))

        15.0412333333 0.377670242702

In [6]: under9 = [triple for triple in aggregate if triple[0]<0.9]
        all_con = [i[1] for i in under9]
        print(np.mean(all_con),np.std(all_con))

        0.706273205716 0.184959712835
```

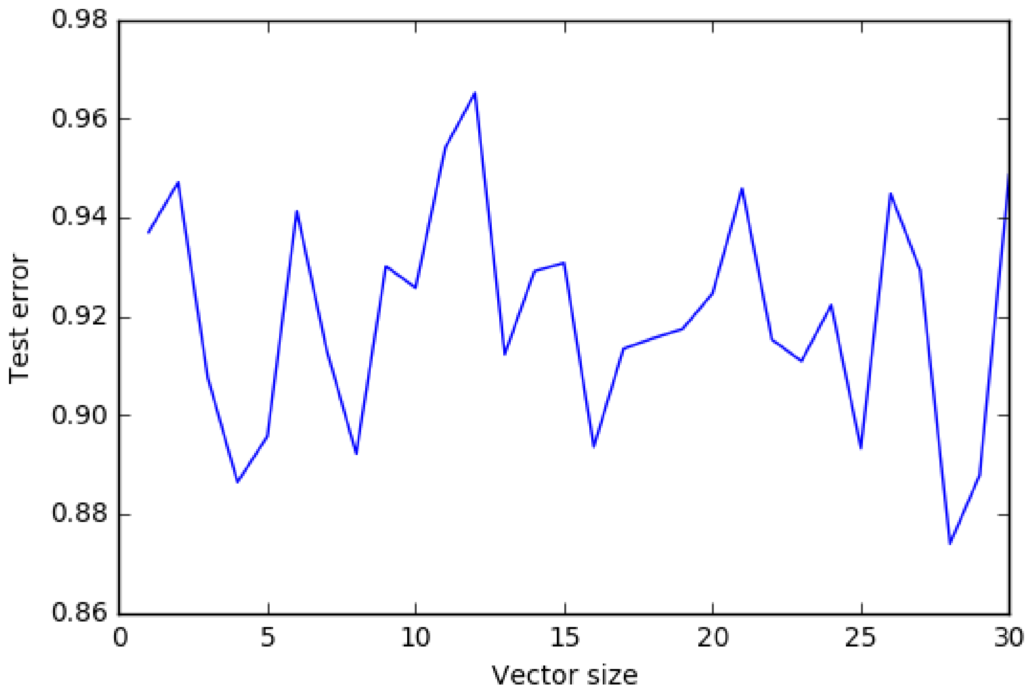
Momentum score calculation for HMF



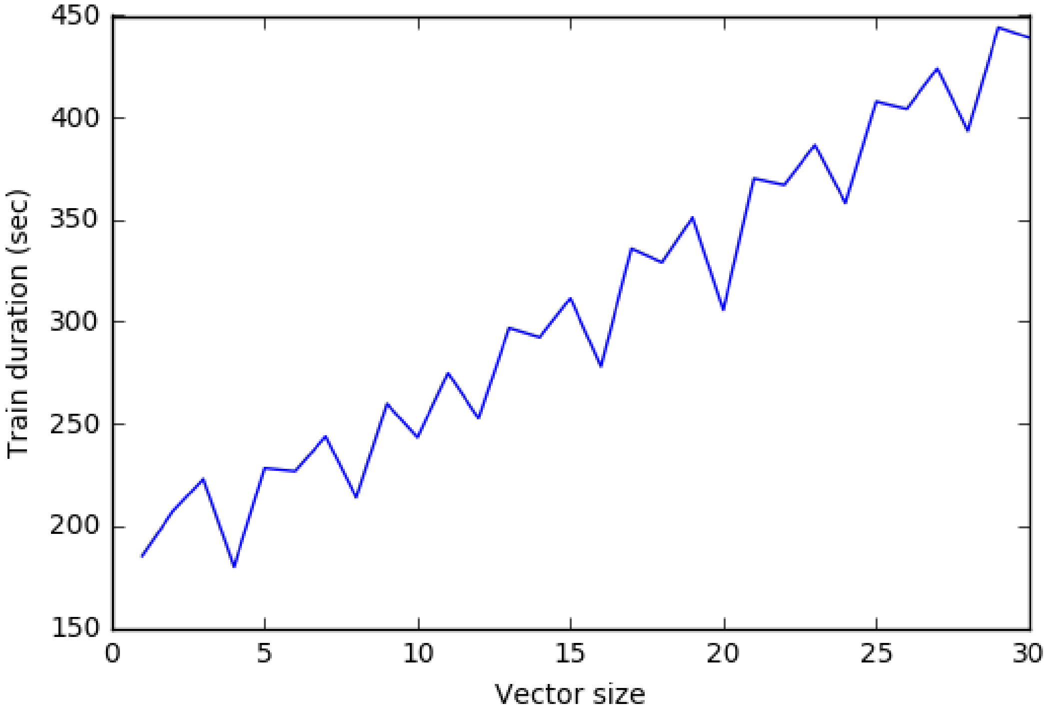
Step Plot to see performance of dimension split



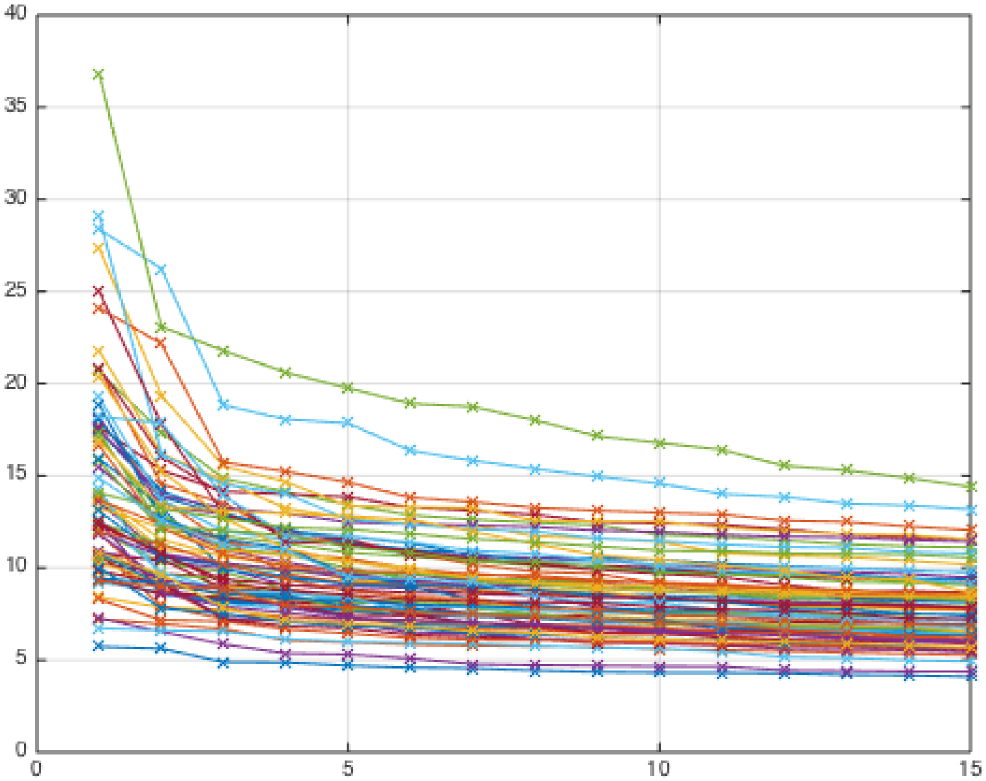
Duration of training each slit according to steps



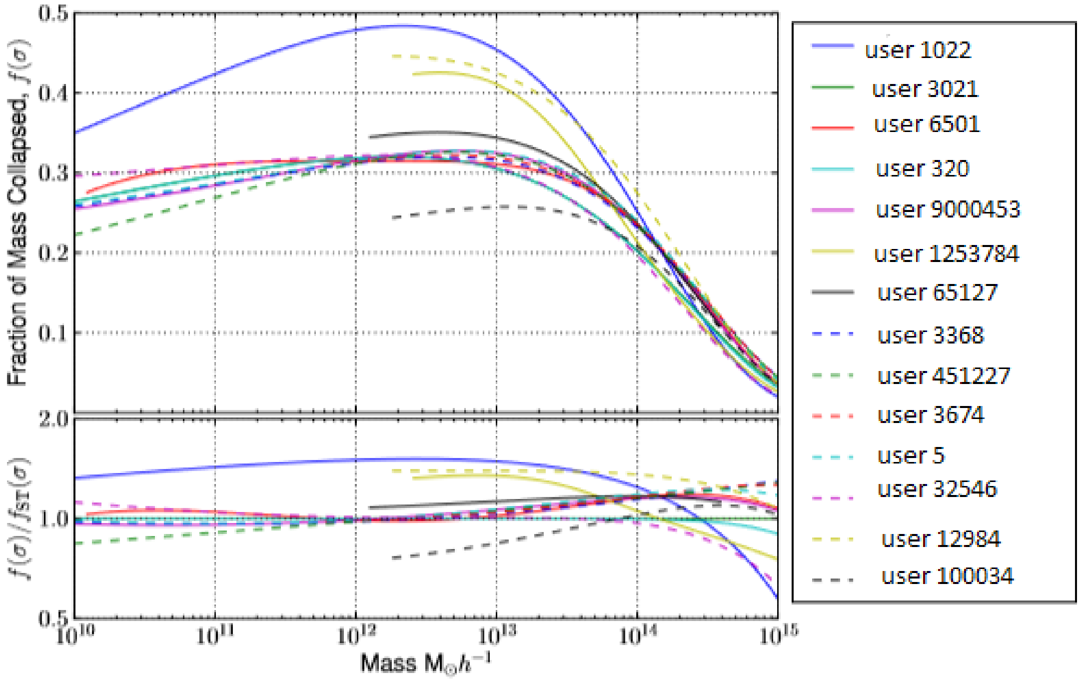
Test error plot for small data fragments with insufficient input



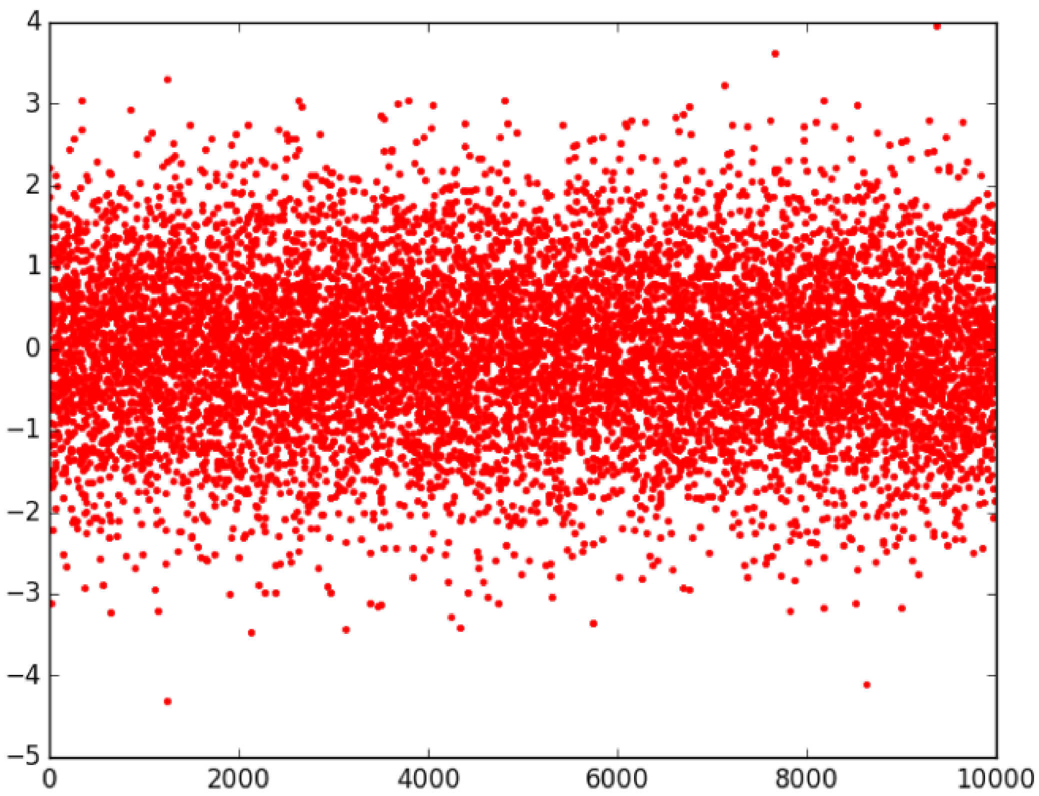
Train Duration Plot for small fragments



Plot processing steps of negative nodes of head()



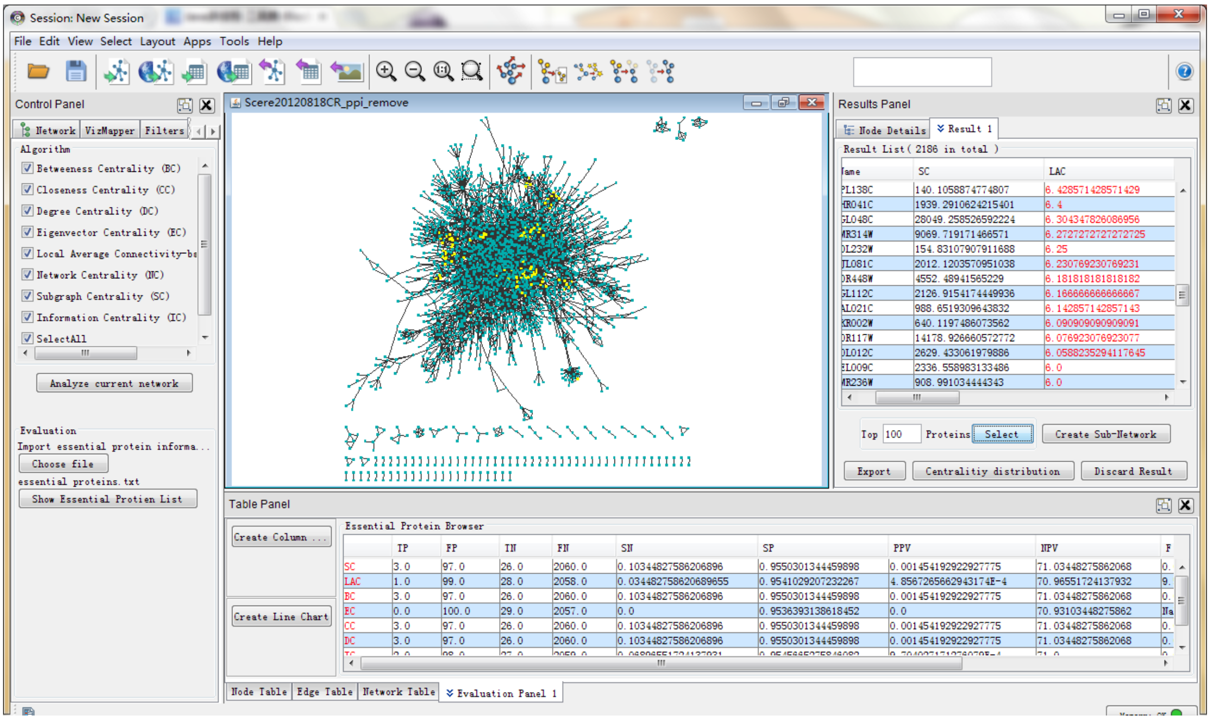
Halo mass weight calculation plot



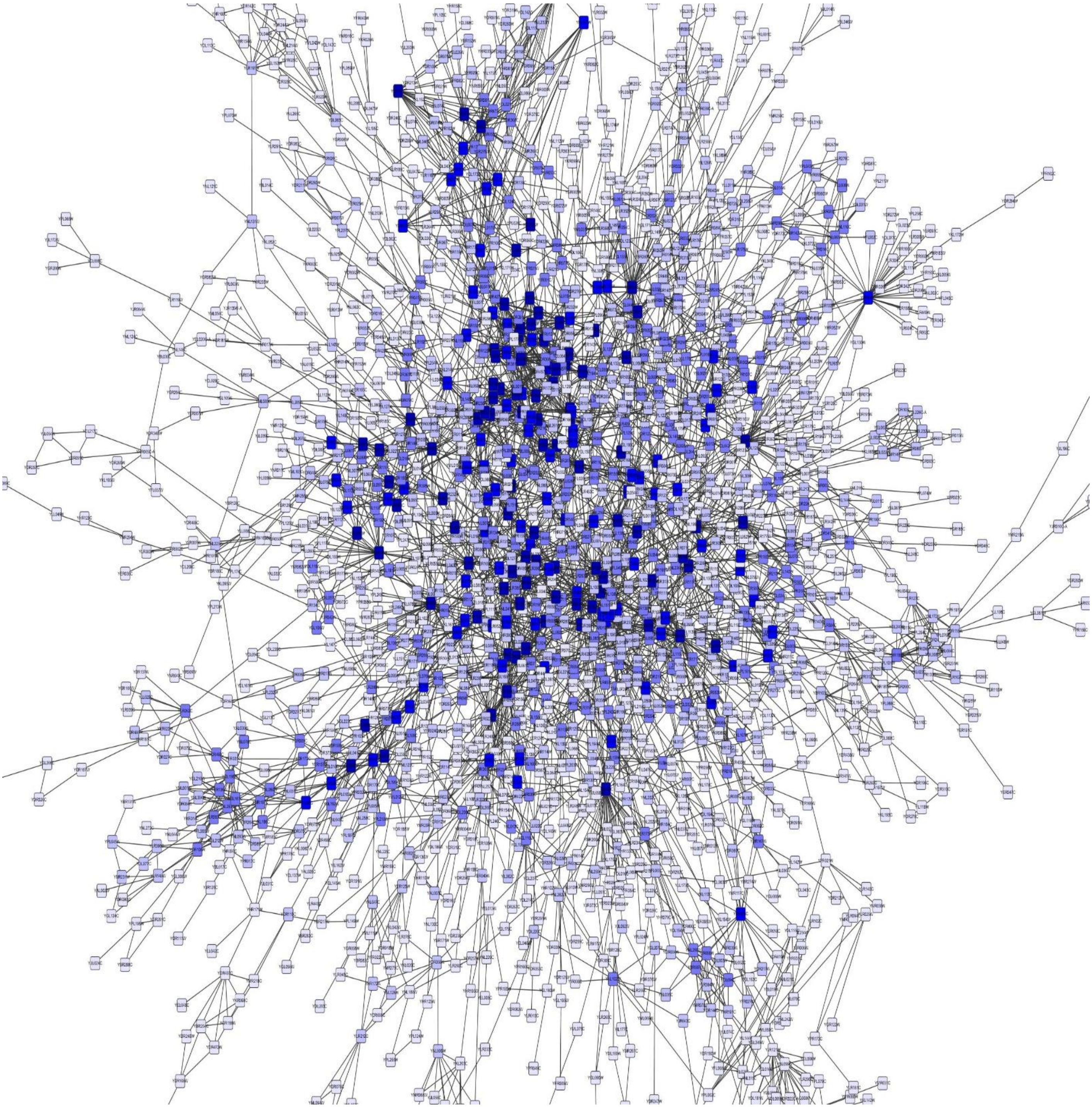
Nodes without incoming or outgoing edges

title	features	hashValues	distCol
Bridgeport (disam...	(179144,[0,1,55,7...	[61999.0,7204.0,1...	0.75
Sedalia	(179144,[0,1,3,7,...	[61999.0,7204.0,1...	0.8
Glenmont	(179144,[0,1,3,7,...	[61999.0,7204.0,1...	0.8
Medway (disambigu...	(179144,[0,1,7,55...	[61999.0,7204.0,1...	0.8
Farmville	(179144,[0,1,3,7,...	[61999.0,7204.0,1...	0.8
Murfreesboro	(179144,[0,1,3,7,...	[61999.0,7204.0,1...	0.8
Burning Springs	(179144,[0,3,19,2...	[61999.0,7204.0,9...	0.8
Morganton	(179144,[0,1,3,7,...	[61999.0,7204.0,1...	0.8
Richton	(179144,[0,1,3,7,...	[61999.0,7204.0,1...	0.8
Mechanicsburg	(179144,[0,1,3,7,...	[61999.0,7204.0,1...	0.8
Waynesboro	(179144,[0,1,3,7,...	[61999.0,7204.0,1...	0.8
Big Run	(179144,[0,3,10,2...	[49107.0,7204.0,1...	0.8
Malden	(179144,[0,1,7,55...	[37855.0,7204.0,1...	0.8
Lewistown	(179144,[0,1,3,7,...	[61999.0,7204.0,1...	0.8
Blacksburg	(179144,[0,1,3,7,...	[61999.0,7204.0,1...	0.8
Frankfort	(179144,[0,1,55,7...	[55850.0,7204.0,1...	0.8
Rockford	(179144,[76,92,68...	[32399.0,73373.0,...	0.8
Court of Appeals ...	(179144,[0,1,55,7...	[59866.0,7204.0,1...	0.8
Lindenhurst	(179144,[0,1,3,7,...	[61999.0,7204.0,1...	0.8
Shepherdstown	(179144,[0,1,3,7,...	[61999.0,7204.0,1...	0.8

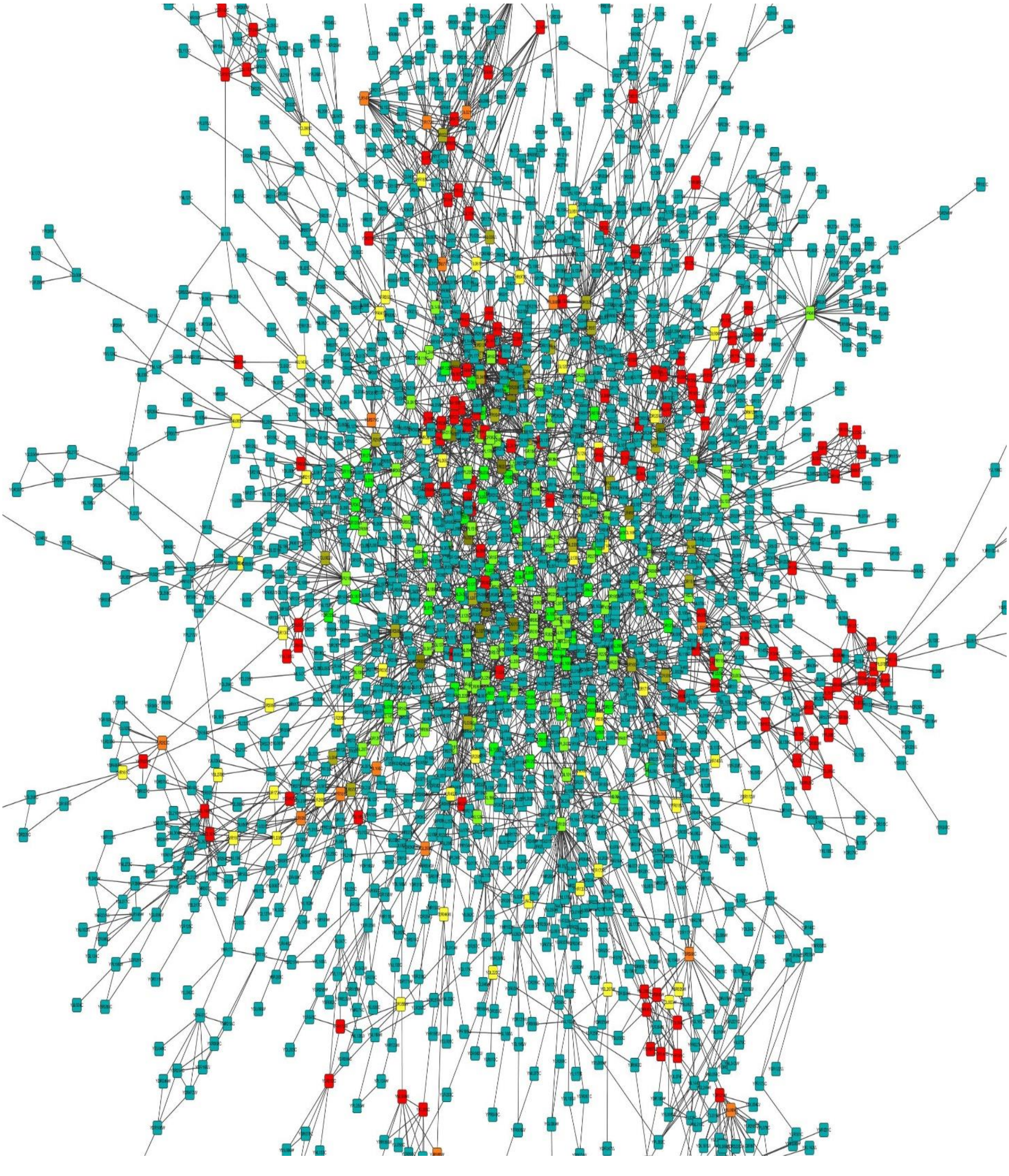
In progress halo mass text miner



Processed graph



Mined nodes in Graphical visuals



Clustered nodes with connections and removals

Conclusion

We took IMDB dataset with movie reviews along with user data. Mined them using several data cleaning process. Implementation of newly constructed Halo mass function followed by several visualisations were done to see the underlying changes. An internal visualizer called **Cyspo** is used to generate visual property of graphs we are dealing with.

It clearly depicts clusters and nodes without negative edge. And the whole system is under process to be implemented along with standard recommender machines.

Our primary goal is achieved hereby. Citations has been sent to different bodies for review. We are expecting positive outcome from reviews.

References

1. Ade, P. A. R., Aghanim, N., Armitage-Caplan, C., Arnaud, M., Ashdown, M., Atrio-Barandela, F., Aumont, J., Baccigalupi, C., Banday, A. J., et al., Mar. 2013. Planck 2013 results.
2. XVI. Cosmological parameters. Angulo, R. E., Springel, V., White, S. D. M., Jenkins, A., Baugh, C. M., Frenk, C. S., Nov. 2012.
3. Scaling relations for galaxy clusters in the Millennium-XXL simulation. MNRAS 426, 2046–2062. Bagla, J. S., Ray, S., Apr. 2005.
4. Comments on the size of the simulation box in cosmological N-body simulations. MNRAS 358, 1076– 1082.
5. Benson, A. J., Farahi, A., Cole, S., Moustakas, L. A., Jenkins, A., Lovell, M., Kennedy, R., Helly, J., Frenk, C., Jan. 2013.
6. Dark matter halo merger histories beyond cold dark matter - I. Methods and application to warm dark matter. MNRAS 428, 1774–1789.
7. Bergstr¨om, L., May 2000. Non-baryonic dark matter: observational evidence and detection methods. Reports on Progress in Physics 63 (5), 793–841. Berlind, A. A., Weinberg, D. H., Aug. 2002.
8. The Halo Occupation Distribution: Toward an Empirical Determination of the Relation between Galaxies and Mass. The Astrophysical Journal 575 (2), 587–616. Bhattacharya, S., Heitmann, K., White, M., Luki´c, Z., Wagner, C., Habib, S., May 2011.
9. MASS FUNCTION PREDICTIONS BEYOND ΛCDM. The Astrophysical Journal 732 (2), 122. Bode, P., Ostriker, J. P., Turok, N., Jul. 2001. Halo Formation in Warm Dark Matter Models. The Astrophysical Journal 556 (1), 93–107.
10. Bond, J. R., Cole, S., Efstathiou, G. P., Kaiser, N., Oct. 1991.
11. Excursion set mass functions for hierarchical Gaussian fluctuations. The Astrophysical Journal 379, 440.
12. Courtin, J., Rasera, Y., Alimi, J.-M., Corasaniti, P. S., Boucher, V., Fu¨zfa, a., Oct. 2010. Imprints of dark energy on cosmic structure formation - II.
13. Non-universality of the halo mass function. Monthly Notices of the Royal Astronomical Society 1931, no–no. Crocce, M., Fosalba, P., Castander, F. J., Gaztan˜aga, E., Apr. 2010.
14. Simulating the Universe with MICE: the abundance of massive clusters. Monthly Notices of the Royal Astronomical Society 403 (3), 1353–1367. Driver, S. P. e. a., May 2011.

15. Galaxy and Mass Assembly (GAMA): survey diagnostics and core data release. Monthly Notices of the Royal Astronomical Society 413 (2), 971–995.
16. Francis, M. J., Lewis, G. F., Linder, E. V., Feb. 2009.
17. Halo mass functions in early dark energy cosmologies. Monthly Notices of the Royal Astronomical Society: Letters 393 (1), L31–L35. Hunter, J. D., 2007.
18. Matplotlib: A 2d graphics environment. Computing In Science & Engineering 9 (3), 90–95.
19. Jenkins, A. R., Frenk, C. S., White, S. D. M., Colberg, J. M., Cole, S., Evrard, a. E., Couchman, H. M. P., Yoshida, N., Feb. 2001.
20. The mass function of dark matter haloes. Monthly Notices of the Royal Astronomical Society 321 (2), 372–384.
21. White, S. D. M., 2002. The Mass Function. The Astrophysical Journal Supplement Series 143, 241–255.
22. Tinker, J., Kravtsov, A. V., 2008. Toward a halo mass function for precision cosmology: the limits of universality. The Astrophysical Journal 688, 709–728.