Abstract

In this article we will explore the model of Latent Dirichlet Allocation theoretically by introducing the model and multiple algorithms for model selection and inference and practically by implementing an inference algorithm based on Gibb's sampling and exploring and visualizing the results. The first section will give an overview about the model and the domain of problems it is applied to. The second section explains a way to implement the model-selection algorithm in Python and a simple approach to parallelize the inference process. In the third section we will train a model on a subset of the simple english Wikipedia and evaluate the results by visualizing the learned topics with the Python library pyLDAviz.

1 Latent Dirichlet Allocation

The model of Latent Dirichlet Allocation (LDA) is a generative probabilistic model for collections of discrete data proposed by Blei, Ng and Jordan [1]. It is a mixture model that tries to model latent topics or concepts of multinomial observations, e.g. words in text corpus.

K : Number of topics to be found by the model

M : Number of documents in the corpus

V : Number of unique terms in the dictionary

term t

$\vec{\alpha} \in \mathbb{R}^K$: Hyperparameter of the document-topic Dirichlet distribution

$\vec{\beta} \in \mathbb{R}^V$: Hyperparameter of the topic-word Dirichlet distribution

$\vec{\vartheta}_m \in \mathbb{R}^K$: Topic distribution of each document m.

$\vec{\phi}_k \in \mathbb{R}^V$: Term distribution of each topic k.

N_m : The length of the m-th document.

$z_{m,n}$: Topic index for the n-th word in document m.

$w_{m,n}$: the n-th word in document m.

Figure 1: Notation

1.1 Generative Process

To understand the model we analyze the LDA generative model, of which an implementation is in the appendix 5.4. To generate a corpus consisting of multiple documents, the steps shown in figure 2 have to be taken.
Given the number of topics K, the number of documents M, the Dirichlet-hyperparameters α, β and the moment of the Poisson distribution ξ we start by specifiying the topics by sampling a categorical topic-word distribution ϕ_k for every topic. These hold a probability for every possible term to occur in the context of the topic. We sample the topic-categoricals from a Dirichlet distribution with the parameter β.
With the Dirichlet-samples we can start to generate documents. For every document d we sample from a Dirichlet distribution again, this time using with the hyperparameter α. The resulting categorical distribution sets probabilities for all topics to be included in the documents. Furthermore we sample a document length from the Poisson distribution.
Finally, for every word to be generated, we sample a topic index from the documents associated document-topic-categorical distribution and then sample the term given the topics multinomial distribution.

Data: Number of topics K, Number of Documents M, Dirichlet parameters α and β, Poisson parameter ξ
Result: Corpus c
for *all topics* $k \in [1, K]$ **do**
  sample topic-term distribution parameters $\phi_k \sim Dir(\beta)$
end
for *all documents* $m \in [1, M]$ **do**
  sample document-topic distribution parameters $\vartheta \sim Dir(\alpha)$
  sample document-length $l_m \sim Poisson(\xi)$
  for *all words* $w \in [1, l_m]$ **do**
    sample topic index $z_{m,n} \sim Mult(\xi)$
    sample term for word $w_{m,n}$ from $p(w_{m,n} \mid z_{m,n}, \beta)$
    $w_{m,n} \sim Mult\left(\phi_{z_{m,n}}\right)$
  end
end

Figure 2: The generative model of LDA [1, 2]

1.2 Inference Algorithm

Various algorithms to tackle the inference problem for LDA have been proposed [1]. In this report we will focus on a specific Markov-Chain Monte-Carlo based method, namely a Gibb's sampler. We will infere with the algorithm explained by Pefta et. al. of which an overview can be found in figure 3.
We will use the following formulas to compute the multinomial parameters given the How to a

2 Implementation

Although the inference algorithm can potentially be parallelized in multiple ways [?, ?], we aim to create a serial implementation for to strive for correctness instead of performance.

2.1 Preparation

First we have to choose a text corpus to estimate the parameters of the LDA-model from. For this we use an instance of the simple english Wikipedia [3], the version

Data: Word vectors $\{\vec{w}\}$, Number of Documents M, Dirichlet parameters α and β, Number of topics T
Result: topic associations $\{\vec{z}\}$
Initialize count variables $n_m^{(k)}, n_m, n_k^{(t)}, n_k$
Initialize multinomial parameters p_t **for** *all documents* $m \in [1, M]$ **do**
  for *all wordpairs* $(i, c) \in m$ **do**
    for *all occurences* c *of a term* i **do**
      sample topic index $z_{m,n} = k \sim Mult\,(1/K)$
      increment document-count: $n_m^{(k)} + = 1$
      increment document-sum: $n_m + = 1$
      increment topic-count: $n_k^{(t)} + = 1$
      increment topic-sum: $n_k + = 1$
    end
  end
  Gibb's sampling:
  for *all iterations* **do**
    $n_m^{(k)} - = 1; n_m - = 1; n_k^{(t)} - = 1; n_k - = 1;$
    sample $z_{m,n} = \overline{k} \sim p\,(z_i \mid \vec{z}_{\neg i})$
    $n_m^{(\overline{k})} - = 1; n_m - = 1; n_{\overline{k}}^{(t)} - = 1; n_k - = 1;$
  end
end

Figure 3: A variational inference algorithm [2]

'All pages, current versions only'.
For the purpose of speeding up the development process we will perform our operations on an even smaller subset of only 5 articles, to avoid long loading times on every run. We split some articles of the downloaded file into a smaller file with the script in section 5.1.

2.2 Class : Dataset

We aggregate all operations regarding the preprocessing of the corpus in a class named Dataset, which can be examined in section 5.2. Our goal is to load and parse the XML-file, select the relevant parts of each article, apply linguistic preprocessing to the articles, count the occuring words and build a dictionary. After the processing by our class we want to have a list of lists of pairs (t, c), where t is a term-index and c is the count of instances of a term in a document, as well as a dictionary with all terms.

Parsing XML To parse the XML-file we add the member-function `loadXMLFile` and utilize the the native Python `xml` module, that allows us to extract the articles with just one expression.

Linguistic Preprocessing To prepare the text we use the function `preprocess_string` from the package `gensim` as it does performs all needed operations in one step. As this function can be applied to every document independently it qualifies to be executed in parallel. We exploit this using the `multiprocessing` package. With using `preprocess_string` we perform the following operations:

1. Remove the syntactic constructs of the Wikipedia markup language
2. Strip punctuation
3. Remove multiple whitespaces
4. Drop numeric words
5. Remove stopwords
6. Strip words shorter than three characters
7. Apply stemming

Building a dictionary As a dictionary we use the class `dictionary` from the `gensim`-submodule `corpora.dictionary` which provides us the possibility to efficiently build a dictionary and map from words to the term-index and back utilizing Python-dictionaries.

Counting words As final step we count the occurences of terms in a document and construct the list of lists of pairs. This decreases the size of the in memory significantly and can be speed up by using `multiprocessing` again. Furthermore we choose to drop all Documents with less than four terms.

Utilities For convenience we add functions to load and restore a dataset to Python-serialization format `pickle`, to avoid redoing the previous steps on every startup.

2.3 Class : LDA

This class holds two verisions of the inference algorithm

2.3.1 Serial Algorithm

The serial infernce algorithm from section 1.2 is implemented in the function `fit`.

2.3.2 Simple Parallelization

To parallelize the inference process we introduce the function `fitParallel` which only adds the parameter of the number of used processing cores to the signature of `fit`. To allow multiple threads to access the datastructures of the model all of them need to be in the global namespace. Therefore we publish them with the `global` keyword. Note that this is not a good programming practice as manipulating global state from inside an objects member-function strongly violates the principle of encapsulation and is a severe side-effect. Nonetheless, this allows us to speed up the inference process with very low effort and our model is for exploration only and it is not planned to be used in production.
We parallelize the loop over all documents so that in each iteration we process all documents in parallel. Apart from making the structures publicly available we have to ensure that access to the count variables is locked. We decrease the document-topic- and topic-term counts, as well as the corresponding sum-structures before sampling a new topic index and increase them afterwards. In this, the topic-term structures could possibly accessed by multiple threads at once. This can lead to race conditions where threads read the wrong counts which leads again to a faulty

parameter estimation. We avoid this by introducing locks for terms and topics, that ensure unique access.

3 Evaluation

To evaluate the topics that the model has learned in the training process on the wikipedia dataset we will use the Python package PyLDAviz. All plots show an intertopic distance map on the left side and a histogram on the right side. The distance map is a two dimensional projection of the topic space computed with a multidimensional scaling algorithm [4]. The Histogram shows the most frequent terms for the chosen topic in descending order, in which the the red bar shows the term frequency in the topic and the blue bar the overall term frequency. We will look at the results of a model trained with the parallel algorithm..

3.1 Performance

The measurements where taken on a machine with four AMD Opteron 6174 processors with 12 cores each.

Dataset Parsing the full XML-file and preprocessing the text took 126.12 seconds and computing the term-count pairs took 126.5174 seconds resulting in a corpus with 144337 documents and 421397 terms. Storing the corpus serialized consumes about 127 mb.

Inference The model in use is trained for 500 iterations on a subset of the whole corpus of 196934 documents with 752612 unique terms which took about 10 hours.

4 Learned Topics

Finally we can evalute the learned topics, plots created with `pyLDAvis` are included figures 4, 5 and 6.

Topic 1 contains many terms related to the governmental activities of countries and yearly statics of economy. Representative terms are "population","gdp","year","rank" and "establish".

Topic 2 consists of terms related to internet, communcation and publication. The five most frequent terms are “http”, “title”,“url”,“web” and “cite”.

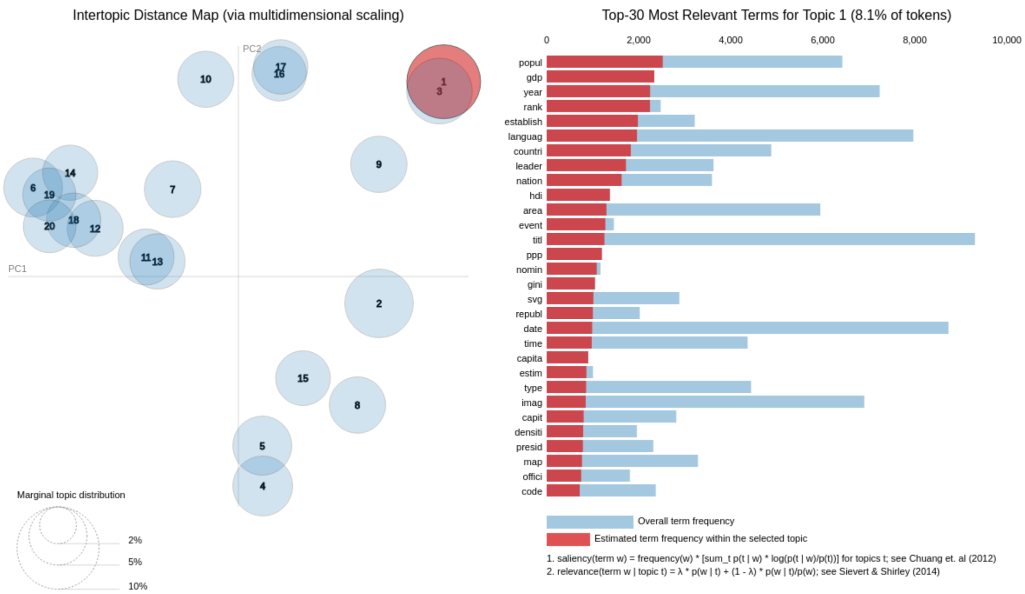
Topic 3 intersects with the first topic in the distance map. Indeed it shares the stems of “population”, “government”, “leader” and “capital” but lacks terms related to the economy.

Topic 4 seems to be related to american motion picture industry and actors. Frequent terms are “movie”,“american”, “award” and “best”. Less frequent are “season”,the stem of “series” and “star”.

Topic 5 is related to rock music with “music”, “rock” and “record” as most frequent terms and to the makings of a band in terms of personnel with “group”, “member” and “singer”. In the distance map we can see a possible intersection, at least a low relative distance to the fourth topic. This makes sense as both topics are related to entertainment, art and public media.

Topic 6 has “people”, “word”, “mean”,“differ” and “like” and “us” as most frequent terms. This points to communcation of intentions between people and evaluation.

(a) Topic 1



(b) Topic 2

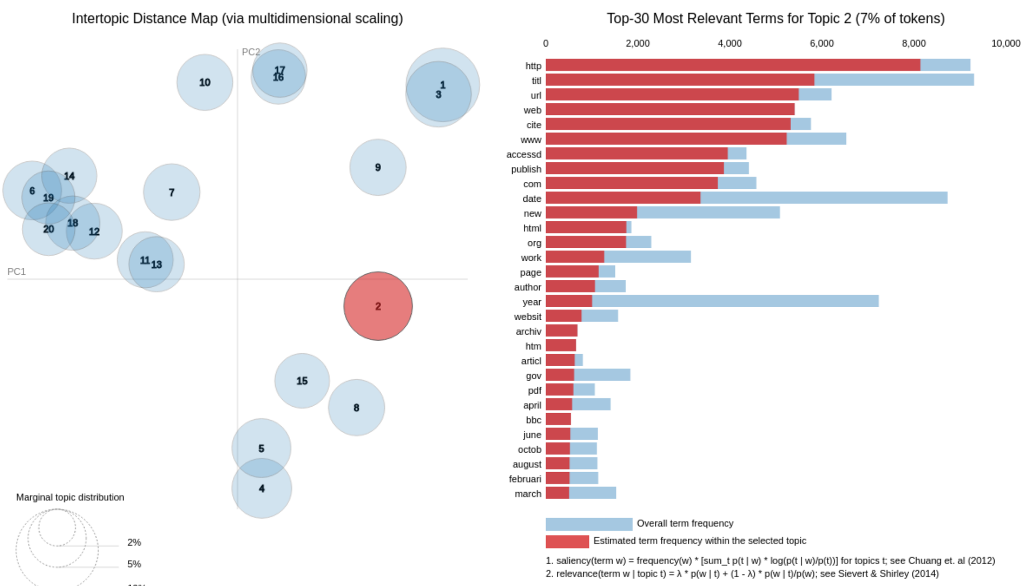
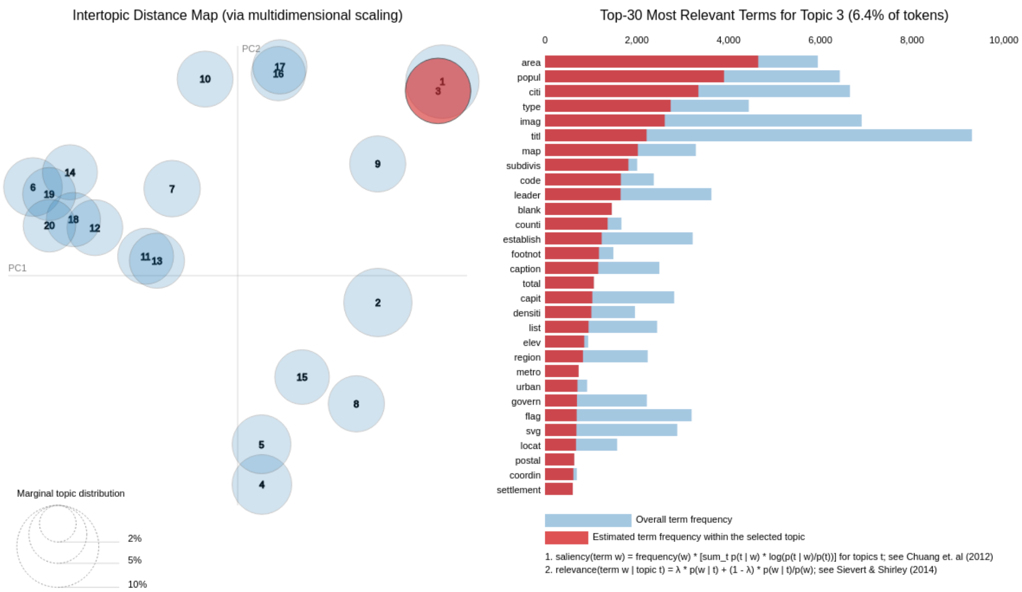


Figure 4: Plots of the first six topics

(a) Topic 3



(b) Topic 4

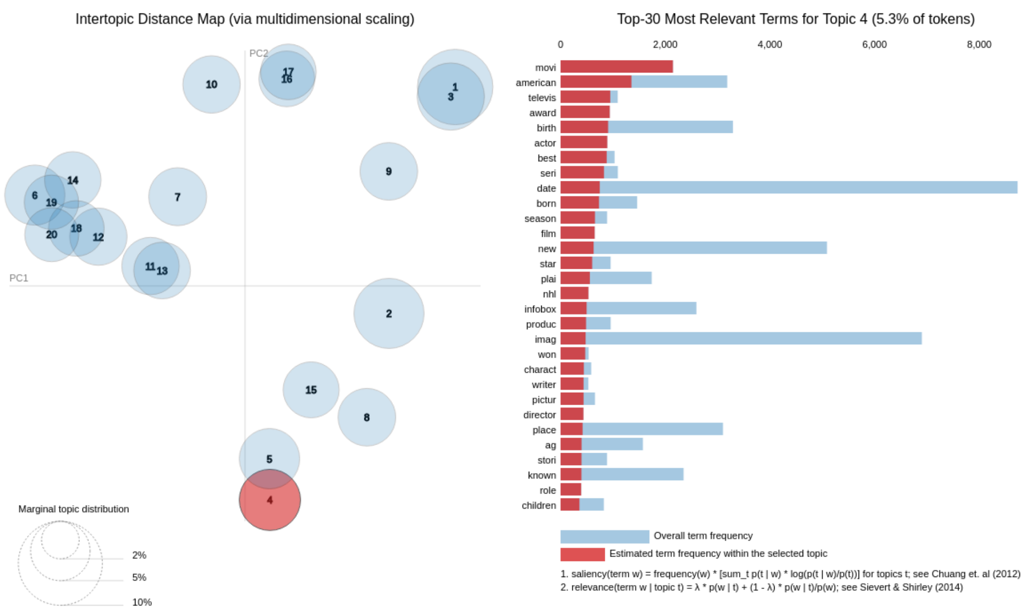
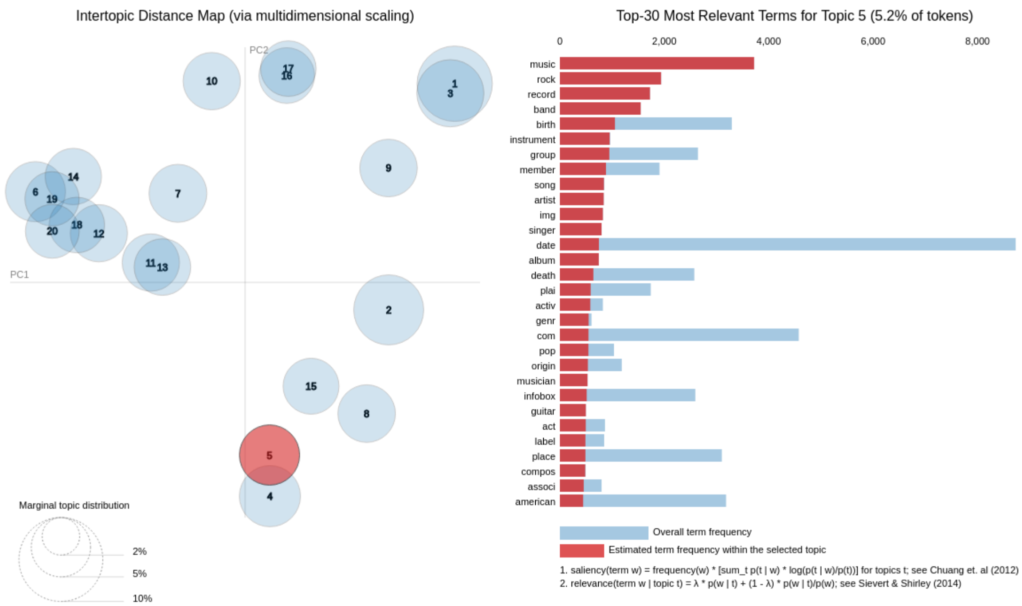


Figure 5: Plots of topic 3 and 4

(a) Topic 5



(b) Topic 6

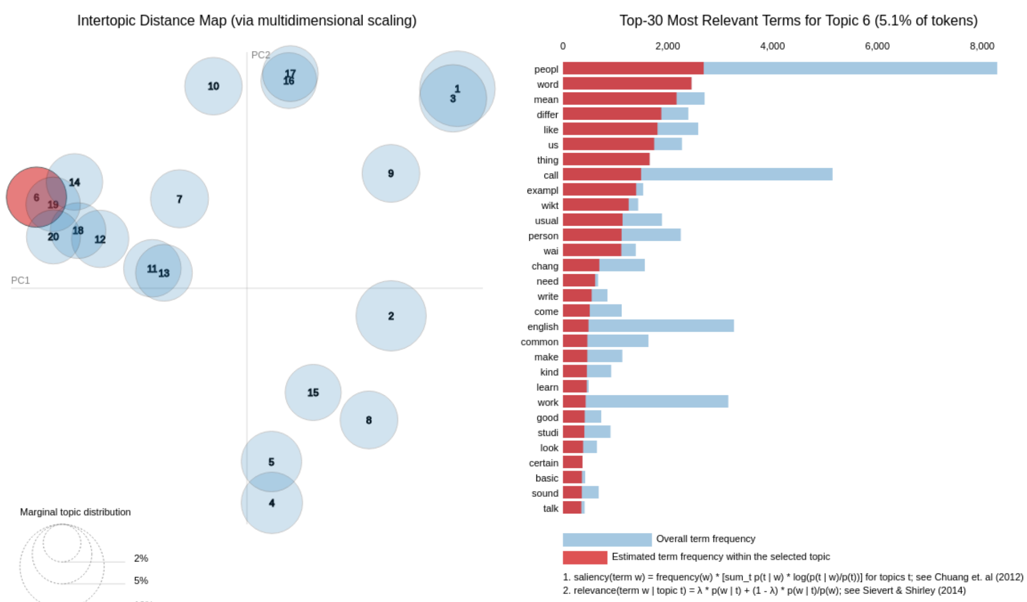


Figure 6: Plots of the first six topics

References

[1] D. M. Blei, A. Y. Ng, M. I. Jordan, and J. Lafferty, "Latent dirichlet allocation," 2003.

[2] Heinrich, "Parameter estimation for text analysis," 2005.

[3] "simplewiki dump progress on 20181120," https://dumps.wikimedia.org/simplewiki/20181120/, (Accessed on 12/15/2018).

[4] S. Sievert, "Ldavis: A method for visualizing and interpreting topics," 2014.

5 Appendix

5.1 Extract subset

```
#!/usr/bin/env bash

cd dataset

#Choose how many pages to extract
n=8000
#Get line number of n-th occurence of the closing tag "</page>"
lineNum=$(
  grep -n  "</page>" simplewiki-20181120-pages-meta-current.xml | \
    head -n${n} | \
    tail -n1 | \
    cut -d: -f1 \
)

#Copy large set upto lineNum into the file small.xml
head simplewiki-20181120-pages-meta-current.xml -n${lineNum} > small.xml

#Add closing tag to small.xml
echo "</mediawiki>" >> small.xml
```

5.2 Class : Dataset

```
import xml.etree.ElementTree as ET
import multiprocessing as mp
import gensim as gsm
import scipy.sparse as sps
import scipy as sp
import numpy as np
from functools import partial
import time
from tqdm import tqdm
import pickle
import sys


def preprocessText(page, onlyOverview=True):
    # Shortcut to get it running
    # return page
    return gsm.parsing.preprocessing.preprocess_string(page)
```

```
def tuples2Matrix(tuples,
                  dictionarySize,
                  matrixFormat,
                  matrixDType):
    matrix = matrixFormat(
        (1, dictionarySize),
        dtype=matrixDType
    )

    for t in tuples:
        matrix[0, t[0]] = t[1]

    return matrix


class DataSet:
    def __init__(self,
                 path='../dataset/small.xml',
                 verbose=True,
                 filterDocumentLengthSmallerThan=4
                 ):
        self.verbose = verbose
        self.filterDocs = filterDocumentLengthSmallerThan

    def load(self, path='../dataset/small.xml'):
        self.path = path

        if self.verbose:
            print("Dataset => Loading XML File")

        start = time.perf_counter()
        documents, self.documentLengths, self.dictionary = self.loadXMLFile(
            path)
        end = time.perf_counter()
        self.loadTime = end - start

        if self.verbose:
            print("Dataset => Parsing " + str(len(documents)) +
                  " documents took: " + "{:10.4f}".format(self.loadTime) + "s")

        if self.verbose:
            print("Dataset => Counting Terms")

        start = time.perf_counter()
        self.documents = self.countTerms(documents, self.dictionary)
```

```
        self.documents = [
            d for d in self.documents if len(d) > self.filterDocs
        ]

        self.docLengths = [
            int(
                np.sum(
                    [pair[1] for pair in doc]
                )
            )
            for doc in self.documents
        ]

        self.termCounts = np.ones(len(self.dictionary))
        for document in self.documents:
            for termIndex, count in document:
                self.termCounts[termIndex] += count

        end = time.perf_counter()
        self.countTermsTime = end - start

        if self.verbose:
            print("Dataset => Building took: {:10.4f}s".format(
                self.countTermsTime))
            print("Dataset => {:10} documents and {:10} terms".format(
                self.numOfDocuments(), self.dictionarySize()))

    def numOfDocuments(self):
        return len(self.documents)

    def documentLengths(self):
        return self.docLengths

    def dictionarySize(self):
        return len(self.dictionary)

    def countTerms(self, documents, dictionary):
        with mp.Pool(mp.cpu_count() - 1) as p:
            counts = p.map(dictionary.doc2bow, tqdm(
                documents, desc='Counting words'))
        return counts

    def loadXMLFile(self, path):
        documents = list()
        root = ET.parse(path).getroot()
        xmlNamespaces = {'root': 'http://www.mediawiki.org/xml/export-0.10/'}
```

```
        # Extract text-attribute of pages in Wikipedia-namespace '0'
        texts = [
            page.find('root:revision', xmlNamespaces)
            .find('root:text', xmlNamespaces).text
            for page in root.findall('root:page', xmlNamespaces)
            if 0 == int(page.find('root:ns', xmlNamespaces).text)
        ]

        # Only use description text
        texts = [text.split('==')[0] for text in texts]

        # Parallel preprocessing of pages
        with mp.Pool(mp.cpu_count() - 1) as p:
            documents = p.map(preprocessText, tqdm(
                texts, desc='Preprocessing text'))
        documentsLengths = list(map(len, documents))
        # Build gensim dictionary
        dictionary = gsm.corpora.dictionary.Dictionary(documents)
        return [documents, documentsLengths, dictionary]

    def saveToDir(self, savePath):
        with open(savePath + 'corpus.pickle', 'wb') as handle:
            pickle.dump(self.documents, handle,
                        protocol=pickle.HIGHEST_PROTOCOL)

        with open(savePath + 'dictionary.pickle', 'wb') as handle:
            pickle.dump(self.dictionary, handle,
                        protocol=pickle.HIGHEST_PROTOCOL)

    def loadFromDir(self, path):
        self.dictionary = pickle.load(
            open(path + "/" + "dictionary.pickle", 'rb')
        )
        self.documents = pickle.load(open(path + "/" + "corpus.pickle", 'rb'))
        self.docLengths = list(map(
            lambda pairList: np.sum(list(map(
                lambda p: p[1],
                pairList
            ))),
            self.documents
        ))
        self.termCounts = np.ones(len(self.dictionary))
        for document in self.documents:
            for termIndex, count in document:
                self.termCounts[termIndex] += count
```

```
if __name__ == '__main__':
    dataset = DataSet()
    dataset.load(sys.argv[1])
```

5.3 Class : LDA

```
from lda.dataset import DataSet
from tqdm import tqdm
import time
import scipy.sparse as sps
import scipy as sp
import numpy as np
import numpy.random as npr
import scipy.stats as spst
import lda.helpers as hlp
import multiprocessing as mp
import itertools as it
import threading as th
from functools import partial
import json
import multiprocessing as mp
import pickle


class LDA():
    def __init__(self,
                 maxit=3,
                 verbose=True,
                 readOutIterations=10,
                 estimateHyperparameters=True,
                 # Mixture proportions; length = num of topics
                 alpha=None,
                 # Mixture components ; length = num of terms
                 beta=None
                 ):
        self.verbose = verbose
        self.maxit = maxit

        self.alpha = None
        self.beta = None
        self.iterations = 0
        self.converged = False
        self.readOutIterations = readOutIterations
        self.lastReadOut = 0
```

```
        if self.verbose:
            print("LDA-Model => constructed")

    def saveJson(self, file):
        saveDict = {'topic_term_dists': self.phi,
                    'doc_topic_dists': self.topicTerm_count_n_kt,
                    'doc_lengths': dataset.documentLengths(),
                    'vocab': dataset,
                    'term_frequency': data_input['term.frequency']}
        jsonString = json.dumps(my_dictionary)

    def fitParallel(self, dataset,
                    nTopics=5,
                    nCores=mp.cpu_count() - 1):
        self.nTopics = nTopics
        self.dataset = dataset
        if self.alpha == None:
            self.alpha = np.repeat(50 / nTopics, nTopics)
        if self.beta == None:
            self.beta = np.repeat(0.01, dataset.dictionarySize())

        if self.verbose:
            print("LDA-Model => fitting to dataset")
        start = time.perf_counter()

        global alpha
        global beta
        alpha = self.alpha
        beta = self.beta

        global documents
        documents = dataset.documents

        global topicLocks
        topicLocks = [mp.Lock() for i in range(nTopics)]

        global termLocks
        termLocks = [mp.Lock() for i in range(dataset.dictionarySize())]

        # M: Number of documents
        # K: Number of topics
        # V: number of Terms

        global topicAssociations_z
```

```
topicAssociations_z = hlp.sharedMultiMatrix(
    dataset.numOfDocuments(), np.max(dataset.documentLengths()), nTopics
)

# M x K
global documentTopic_count_n_mk
documentTopic_count_n_mk = hlp.sharedZeros(
    dataset.numOfDocuments(),
    nTopics
)

# K x v
global topicTerm_count_n_kt
topicTerm_count_n_kt = hlp.sharedZeros(
    nTopics,
    dataset.dictionarySize()
)

for documentIndex in range(dataset.numOfDocuments()):
    document = dataset.documents[documentIndex]
    wordIndex = 0
    for pair in document:
        termIndex = pair[0]
        for c in range(pair[1]):
            topicIndex = topicAssociations_z[documentIndex][wordIndex]
            documentTopic_count_n_mk[documentIndex,
                                     topicIndex] += 1
            topicTerm_count_n_kt[topicIndex, termIndex] += 1
            wordIndex += 1

# M
global documentTopic_sum_n_m
documentTopic_sum_n_m = np.sum(
    documentTopic_count_n_mk, axis=1)
documentTopic_sum_n_m = hlp.sharedArray(
    documentTopic_sum_n_m)
assert (
    len(documentTopic_sum_n_m.shape) == 1
)
assert (
    documentTopic_sum_n_m.shape[0] == dataset.numOfDocuments()
)

# K
global topicTerm_sum_n_k
topicTerm_sum_n_k = np.sum(topicTerm_count_n_kt, axis=1)
```

```
topicTerm_sum_n_k = hlp.sharedArray(topicTerm_sum_n_k)
assert (
    len(topicTerm_sum_n_k.shape) == 1
)
assert (
    topicTerm_sum_n_k.shape[0] == nTopics
)

# end = time.perf_counter()
end = time.perf_counter()
self.initializazionTime = end - start
if self.verbose:
    print("LDA => Initialization took: {:10.4f}".format(
        self.initializazionTime) + "s")

# ------------------------------------------------ Sampling ------------------------------------------------
if self.verbose:
    print("LDA => fitting to Dataset")

start = time.perf_counter()

global processDocument

def processDocument(
    documentIndex,
    documents=documents,
    topicAssociations_z=topicAssociations_z,
    documentTopic_count_n_mk=documentTopic_count_n_mk,
    topicTerm_count_n_kt=topicTerm_count_n_kt,
    documentTopic_sum_n_m=documentTopic_sum_n_m,
    topicTerm_sum_n_k=topicTerm_sum_n_k,
    beta=beta,
    alpha=alpha,
    nTopics=nTopics,
    termLocks=termLocks,
    topicLocks=topicLocks
):
    document = documents[documentIndex]
    wordIndex = 0
    for pair in document:
        termIndex = pair[0]
        termLocks[termIndex].acquire()

        for c in range(pair[1]):
            previousTopicIndex = topicAssociations_z[documentIndex, wordIndex]
```

```
            # For the current assignment of k to a term t for word w_{m,n}
            topicLocks[previousTopicIndex].acquire()
            documentTopic_count_n_mk[documentIndex,
                                     previousTopicIndex] -= 1
            documentTopic_sum_n_m[documentIndex] -= 1
            topicTerm_count_n_kt[previousTopicIndex,
                                 termIndex] -= 1
            topicTerm_sum_n_k[previousTopicIndex] -= 1

            # multinomial sampling acc. to Eq. 78 (decrements from previous step

            params = np.zeros(nTopics)
            for topicIndex in range(nTopics):
                n = topicTerm_count_n_kt[topicIndex,
                                         termIndex] + beta[termIndex]
                d = topicTerm_sum_n_k[topicIndex] + \
                    beta[termIndex]
                f = documentTopic_count_n_mk[documentIndex,
                                             topicIndex] + alpha[topicIndex]
                params[topicIndex] = (n / d) * f
            topicLocks[previousTopicIndex].release()

            # Scale
            params = np.asarray(params).astype('float64')
            params = params / np.sum(params)

            newTopicIndex = hlp.getIndex(
                spst.multinomial(1, params).rvs()[0])

            topicLocks[newTopicIndex].acquire()
            topicAssociations_z[documentIndex,
                                wordIndex] = newTopicIndex
            # For new assignments of z_{m,n} to the term t for word w_{m,n}
            documentTopic_count_n_mk[documentIndex,
                                     newTopicIndex] += 1
            documentTopic_sum_n_m[documentIndex] += 1
            topicTerm_count_n_kt[newTopicIndex,
                                 termIndex] += 1
            topicTerm_sum_n_k[newTopicIndex] += 1

            topicLocks[newTopicIndex].release()
            wordIndex += 1
        termLocks[termIndex].release()

for iteration in tqdm(range(self.maxit), desc='Sampling: '):
    with mp.Pool(mp.cpu_count() - 1) as p:
```

```
                p.map(processDocument, range(len(dataset.documents)))

            self.iterations += 1

            if self.converged and self.lastReadOut > self.readOutIterations:
                print("reading")

            if self.iterations > self.maxit:
                self.converged = True
                if self.verbose:
                    print("LDA.fit() => Maximum number of iterations reached!")

        self.topicAssociations_z = topicAssociations_z
        self.documentTopic_count_n_mk = documentTopic_count_n_mk
        self.topicTerm_count_n_kt = topicTerm_count_n_kt
        self.documentTopic_sum_n_m = documentTopic_sum_n_m
        self.topicTerm_sum_n_k = topicTerm_sum_n_k

        self.compute_phi()
        self.compute_theta()

        end = time.perf_counter()

        self.inferenceTime = end - start

        if self.verbose:
            print("LDA => Fitting took: {:10.4f}".format(
                self.inferenceTime) + "s")
            print("LDA => Convergence took: {:10.4f}".format(self.iterations))

    def fit(self, dataset,
            nTopics=5):
        self.nTopics = nTopics
        self.dataset = dataset
        if self.alpha == None:
            self.alpha = np.repeat(50 / nTopics, nTopics)
        if self.beta == None:
            self.beta = np.repeat(0.01, dataset.dictionarySize())

        if self.verbose:
            print("LDA-Model => fitting to dataset")
        start = time.perf_counter()

        # M: Number of documents
        # K: Number of topics
        # V: number of Terms
```

```
partialMultilist = partial(hlp.randomMultilist, nTopics=nTopics)
self.topicAssociations_z = list(
    map(partialMultilist, dataset.documentLengths()))

# M x K
self.documentTopic_count_n_mk = np.zeros(
    (dataset.numOfDocuments(),
     nTopics)
)

print("Dsize:", dataset.dictionarySize())
# K x v
self.topicTerm_count_n_kt = np.zeros(
    (nTopics,
     dataset.dictionarySize())
)

for documentIndex in range(dataset.numOfDocuments()):
    document = dataset.documents[documentIndex]
    wordIndex = 0
    for pair in document:
        termIndex = pair[0]
        for c in range(pair[1]):
            topicIndex = self.topicAssociations_z[documentIndex][wordIndex]
            self.documentTopic_count_n_mk[documentIndex,
                                          topicIndex] += 1
            self.topicTerm_count_n_kt[topicIndex, termIndex] += 1
            wordIndex += 1

# M
self.documentTopic_sum_n_m = np.sum(
    self.documentTopic_count_n_mk, axis=1)
assert (
    len(self.documentTopic_sum_n_m.shape) == 1
)
assert (
    self.documentTopic_sum_n_m.shape[0] == dataset.numOfDocuments()
)

# K
self.topicTerm_sum_n_k = np.sum(self.topicTerm_count_n_kt, axis=1)
assert (
    len(self.topicTerm_sum_n_k.shape) == 1
)
assert (
```

```
        self.topicTerm_sum_n_k.shape[0] == nTopics
    )
    # end = time.perf_counter()
    end = time.perf_counter()
    self.initializazionTime = end - start
    if self.verbose:
        print("LDA => Initialization took: {:10.4f}".format(
            self.initializazionTime) + "s")

    # ---------------------------------------- Sampling ----------------------------------------
    if self.verbose:
        print("LDA => fitting to Dataset")

    start = time.perf_counter()

    for iteration in tqdm(range(self.maxit), desc='Sampling: '):
        for documentIndex in range(len(dataset.documents)):
            document = dataset.documents[documentIndex]
            wordIndex = 0
            for pair in document:
                termIndex = pair[0]
                for c in range(pair[1]):
                    previousTopicIndex = self.topicAssociations_z[documentIndex][wo

                    # For the current assignment of k to a term t for word w_{m,n}
                    self.documentTopic_count_n_mk[documentIndex,
                                                  previousTopicIndex] -= 1
                    self.documentTopic_sum_n_m[documentIndex] -= 1
                    self.topicTerm_count_n_kt[previousTopicIndex,
                                              termIndex] -= 1
                    self.topicTerm_sum_n_k[previousTopicIndex] -= 1

                    # multinomial sampling acc. to Eq. 78 (decrements from previous

                    params = np.zeros(self.nTopics)
                    for topicIndex in range(self.nTopics):
                        n = self.topicTerm_count_n_kt[topicIndex,
                                                      termIndex] + self.beta[termIn
                        d = self.topicTerm_sum_n_k[topicIndex] + \
                            self.beta[termIndex]
                        f = self.documentTopic_count_n_mk[documentIndex,
                                                          topicIndex] + self.alpha[
                        params[topicIndex] = (n / d) * f

                    # Scale
                    params = np.asarray(params).astype('float64')
```

```
                        params = params / np.sum(params)
                        newTopicIndex = hlp.getIndex(
                            spst.multinomial(1, params).rvs()[0])

                        self.topicAssociations_z[documentIndex][wordIndex] = newTopicIn
                        # For new assignments of z_{m,n} to the term t for word w_{m,n}
                        self.documentTopic_count_n_mk[documentIndex,
                                                      newTopicIndex] += 1
                        self.documentTopic_sum_n_m[documentIndex] += 1
                        self.topicTerm_count_n_kt[newTopicIndex,
                                                  termIndex] += 1
                        self.topicTerm_sum_n_k[newTopicIndex] += 1
                        wordIndex += 1

            self.iterations += 1

            if self.converged and self.lastReadOut > self.readOutIterations:
                print("reading")

            if self.iterations > self.maxit:
                self.converged = True
                if self.verbose:
                    print("LDA.fit() => Maximum number of iterations reached!")

        self.compute_phi()
        self.compute_theta()

        end = time.perf_counter()

        self.inferenceTime = end - start

        if self.verbose:
            print("LDA => Fitting took: {:10.4f}".format(
                self.inferenceTime) + "s")
            print("LDA => Convergence took: {:10.4f}".format(self.iterations))

    def saveToDir(self, savePath, protocol=2):
        # Safe topic_term_dists, doc_topic_dists, doc_lengths, vocab, term_frequency
        with open(savePath + 'phi.pickle', 'wb') as handle:
            pickle.dump(
                self.phi, handle,
                protocol=protocol
            )

        with open(savePath + 'theta.pickle', 'wb') as handle:
            pickle.dump(
```

```
                self.theta, handle,
                protocol=protocol
            )

        with open(savePath + 'docLengths.pickle', 'wb') as handle:
            pickle.dump(
                self.dataset.docLengths, handle,
                protocol=protocol
            )

        with open(savePath + 'vocabulary.pickle', 'wb') as handle:
            pickle.dump(
                list(map(
                    lambda x: self.dataset.dictionary[x], self.dataset.dictionary.keys(
                protocol=protocol
            )
        with open(savePath + 'termFrequencys.pickle', 'wb') as handle:
            pickle.dump(self.dataset.termCounts, handle,
                        protocol=protocol
                        )

    def compute_phi(self):
        """Calculate Parameters of The topic-term multinomial"""
        self.phi = np.zeros((self.nTopics, self.dataset.dictionarySize()))
        for topicIndex, termIndex in tqdm(it.product(range(self.nTopics), range(self.da
            self.phi[topicIndex, termIndex] = (self.topicTerm_count_n_kt[topicIndex, te
                                              + self.beta[termIndex]) / (self.topicTerm

    def compute_theta(self):
        """Calculate Parameters of The document-topic multinomial"""
        self.theta = np.zeros((self.dataset.numOfDocuments(), self.nTopics))
        for documentIndex, topicIndex in tqdm(it.product(range(self.dataset.numOfDocume
            self.theta[documentIndex, topicIndex] = (self.documentTopic_count_n_mk[docu
                self.documentTopic_sum_n_m[documentIndex] + self.alpha[topicIndex])


if __name__ == '__main__':
    dataset = DataSet()
    model = LDA()
    model.fit(dataset)
```

5.4 Class : LDA

```
import numpy as np

from scipy.stats import poisson
```

```
from scipy.stats import multinomial
from scipy.stats import dirichlet


class GenMod:
    def __init__(self,
                 mDocuments,
                 kTopics,
                 topicWordConcentration_beta,
                 documentTopicConcentration_alpha,
                 poissonMoment=500):
        self.mDocuments = mDocuments
        self.kTopics = kTopics
        self.topicWordDirichlet = dirichlet(topicWordConcentration_beta)
        self.topicWordMultinomialsPhi = self.topicWordDirichlet.rvs(size=self.kTopics)
        self.documentTopicDir = dirichlet(documentTopicConcentration_alpha)
        self.documentTopicMultinomialsTheta = self.documentTopicDir.rvs(mDocuments)
        self.poissonMoment = poissonMoment
        self.docs, self.wordTopicLists = self.randomize()

    def randomize(self):
        docs = list()
        wordTopicLists = list()
        for documentIndex in range(self.mDocuments):
            documentLengths = [poisson.rvs(self.poissonMoment)
                               for i in range(self.mDocuments)]
            doc = list()
            docs.append(doc)

            wordTopicList = list()
            wordTopicLists.append(wordTopicList)

            for wordIndex in range(documentLengths[documentIndex]):
                topicIndex = multinomial.rvs(
                    p=self.documentTopicMultinomialsTheta[documentIndex, :], n=1)
                wordTopicList.append(topicIndex)
                word = multinomial.rvs(
                    p=self.topicWordMultinomialsPhi[topicIndex, :][0], n=1)
                doc.append(word)
        return ([docs, wordTopicLists])
```