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 how to implement the model-selection algorithm and will guide the way to . In the third section we will train the 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

$\vec{\alpha} \in \mathbb{R}^K$: 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

To understand the structure of the model we will look at the Likelihood-function for The probability of a term to occur in a document $p\left(w = t\right)$ is modeled as a marginal distribution of the joint distribution over topics and terms.

$$p\left(\vec{w}_m, \vec{z}_m, \vec{\vartheta}_m \,\middle|\, \vec{\alpha}, \vec{\beta}\right) = \prod_{w_{m,n}}^{N_m} p\left(w_{m,n} \,\middle|\, \vec{\phi}_{z_{m,n}}\right) p\left(z_{m,n} \,\middle|\, \vec{\vartheta}_m\right) p\left(\vec{\vartheta}_m \,\middle|\, \vec{\alpha}\right) p\left(\phi \,\middle|\, \vec{\beta}\right)$$

1.1 Generative Process

To understand the model better we analyze the LDA generative model. To generate a corpus consisting of documents, that consist of words, 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 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 own document-topic-categorical distribution and then sample the term given the
Look in section 4.4 for an implementation of the process.

```
Data: Number of topics $K$, Number of Documents $M$, Dirichlet parameters
      $\alpha$ and $\beta$, Poisson parameter $\xi$
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 $p(w_{m,n} \mid z_{m,n}, \beta)$
         $w_{m,n} \sim Mult(\phi_{z_{m,n}})$
    end
end
```

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

1.2 Model Selection

1.2.1 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 execute the inference with the algorithm explained by Pefta et. al.

2 Imlementation

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 a dump from the simple english wikipedia [3], namely the

version named '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 4.1.

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 4.2. Our goal here is to process all articles and end up with a datatructure that provides us with with counts of terms for every document. Therefore we have to establish a common dictionary containing all occurring words from all documents and process

2.3 Class : LDA

3 Evaluation

To evaluate the topics that the model has learned in the trainingprocess on the wikipedia dataset we will use the Python package PyLDAviz. The plots of the first six found topics can be found in the figures 3, 4 and 5. 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. The model in use is trained for 50 iterations on the first 10000 artices of the dataset consiting of 196934 documents.

To try to find a possible semantic we will take a look at the most frequent terms in the topics.

Topic 1 The first topic contains many terms related to the internet, communcation and publication. The five most frequent terms are “title”,“http”,“cite”,“url” and “web”.

Topic 2

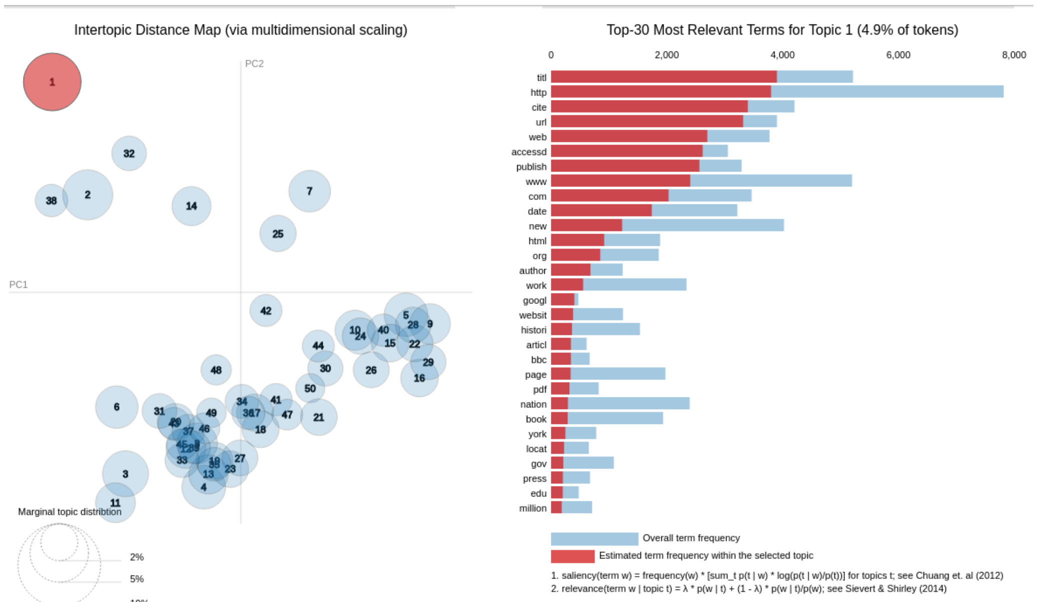
Topic 3

Topic 4

Topic 5

Topic 5

(a) Topic 1



(b) Topic 2

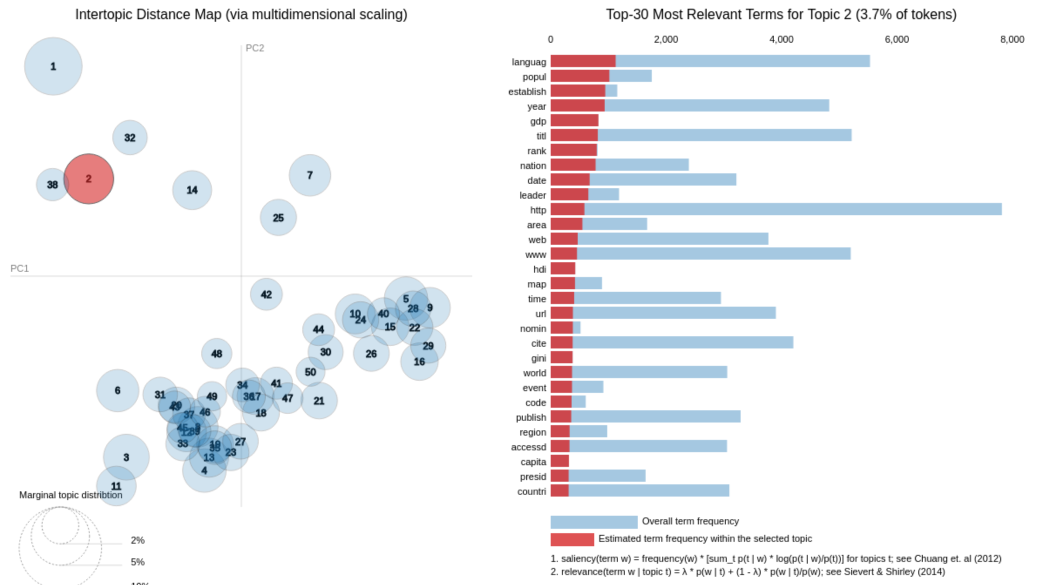
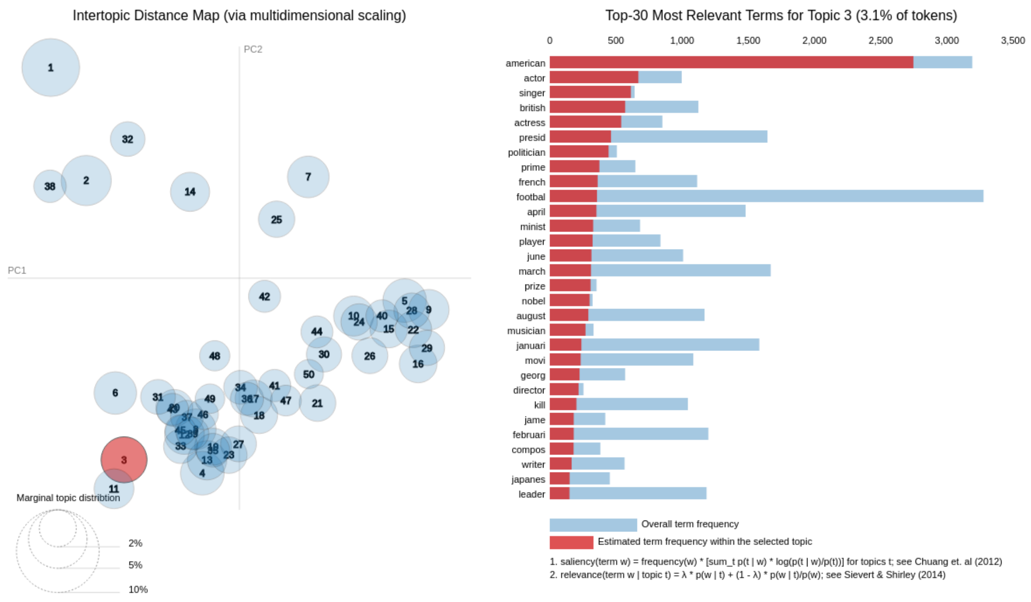


Figure 3: Plots of the first six topics

(a) Topic 3



(b) Topic 4

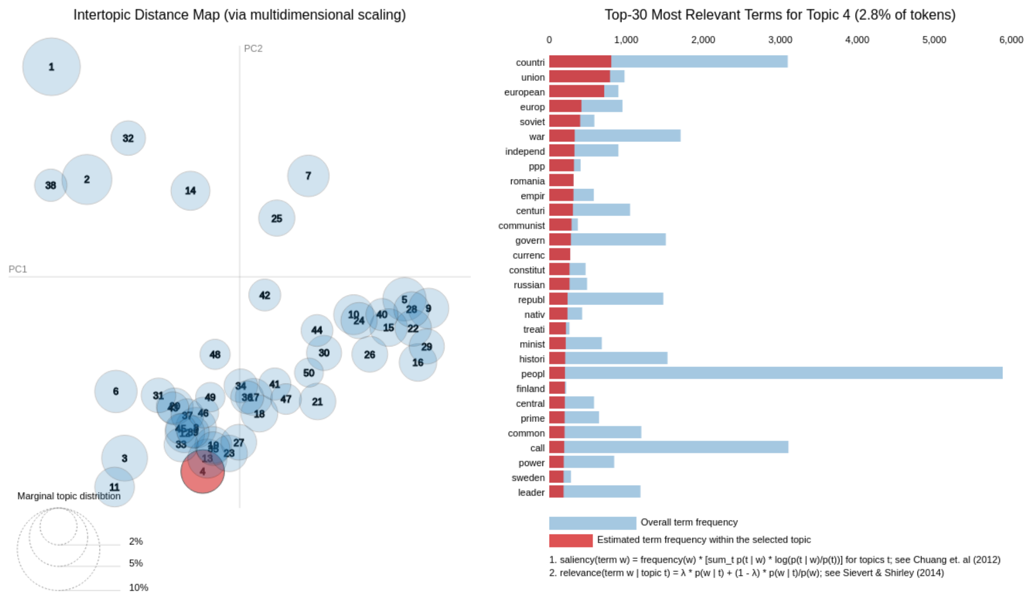
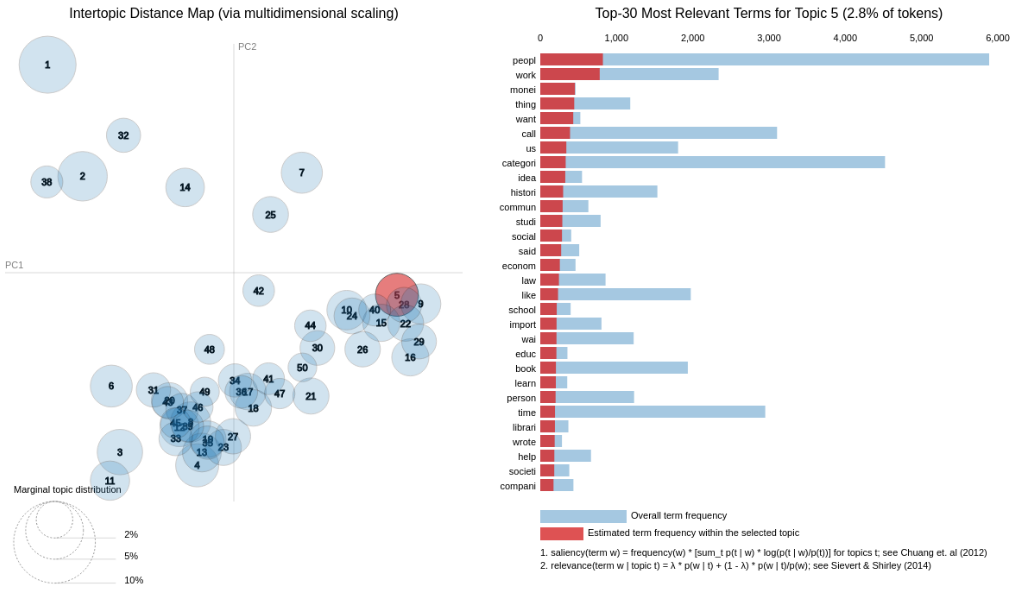


Figure 4: Plots of topic 3 and 4

(a) Topic 5



(b) Topic 6

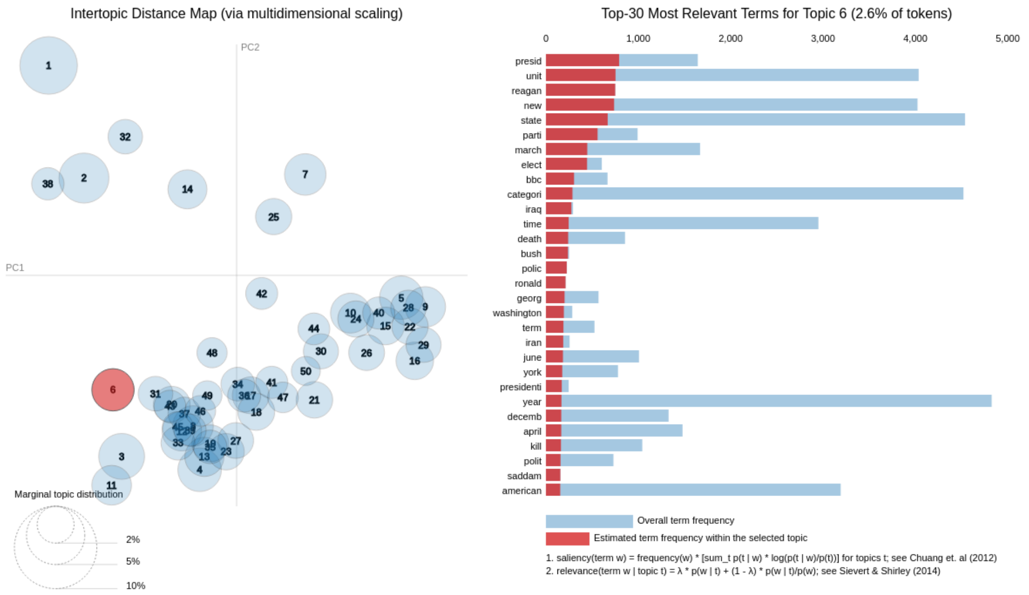


Figure 5: Plots of the first six topics

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.

4 Appendix

4.1 Extract subset

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

cd dataset

#Choose how many pages to extract
n=100
#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.xm

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

4.2 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,
                 ):
        self.verbose = verbose

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

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

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

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

        if self.verbose:
            print("Dataset => Building Matrix")

        start = time.perf_counter()
        self.documents = self.countTerms(documents, self.dictionary)
        self.docLengths = list(map(lambda pairList: np.sum(
            list(map(lambda p: p[1], pairList))), self.documents))
        end = time.perf_counter()

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

        self.countTermsTime = end - start

        if self.verbose:
            print("Dataset => Building took: "
                  + "{:10.4f}".format(self.countTermsTime) + "s")

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

    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-(

        # 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]

        if self.verbose:
            print('Parse xml')

        # 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",
        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])
```

4.3 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 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
                    self.documentTopic_count_n_mk[documentIndex,
                                                   topicIndex] += 1
                    self.topicTerm_count_n_kt[topicIndex, termIndex] +=
                    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.numOfDocumen
        )

        # K
        self.topicTerm_sum_n_k = np.sum(self.topicTerm_count_n_kt, axis
        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[d

                        # For the current assignment of k to a term t f
                        self.documentTopic_count_n_mk[documentIndex,
                                                      previousTopicInde
                        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 (decremen

                        params = np.zeros(self.nTopics)
                        for topicIndex in range(self.nTopics):
                            n = self.topicTerm_count_n_kt[topicIndex,
                                                          termIndex] +
                            d = self.topicTerm_sum_n_k[topicIndex] + \
                                self.beta[termIndex]
                            f = self.documentTopic_count_n_mk[documentI
                                                              topicInde
                            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][wordInd
                    # For new assignments of z_{m,n} to the term t
                    self.documentTopic_count_n_mk[documentIndex,
                                                  newTopicIndex] +=
                    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.readOutIterat
                print("reading")

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

        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.itera

    def saveToDir(self, savePath, protocol=2):
        # Safe topic_term_dists, doc_topic_dists, doc_lengths, vocab, t
        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.
                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
            self.phi[topicIndex, termIndex] = (self.topicTerm_count_n_k
                                                 self.beta[termIndex]) /

    def compute_theta(self):
        """Calculate Parameters of The document-topic multinomial"""
        self.theta = np.zeros((self.dataset.numOfDocuments(), self.nTop
        for documentIndex, topicIndex in tqdm(it.product(range(self.dat
            self.theta[documentIndex, topicIndex] = (self.documentTopic
                self.documentTopic_sum_n_m[documentIndex] + self.alpha[
if __name__ == '__main__':
    dataset = DataSet()
    model = LDA()
    model.fit(dataset)
```

4.4 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(siz
        self.documentTopicDir = dirichlet(documentTopicConcentration_al
        self.documentTopicMultinomialsTheta = self.documentTopicDir.rvs
        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
                wordTopicList.append(topicIndex)
                word = multinomial.rvs(
                    p=self.topicWordMultinomialsPhi[topicIndex, :][0],
                doc.append(word)
        return ([docs, wordTopicLists])
```