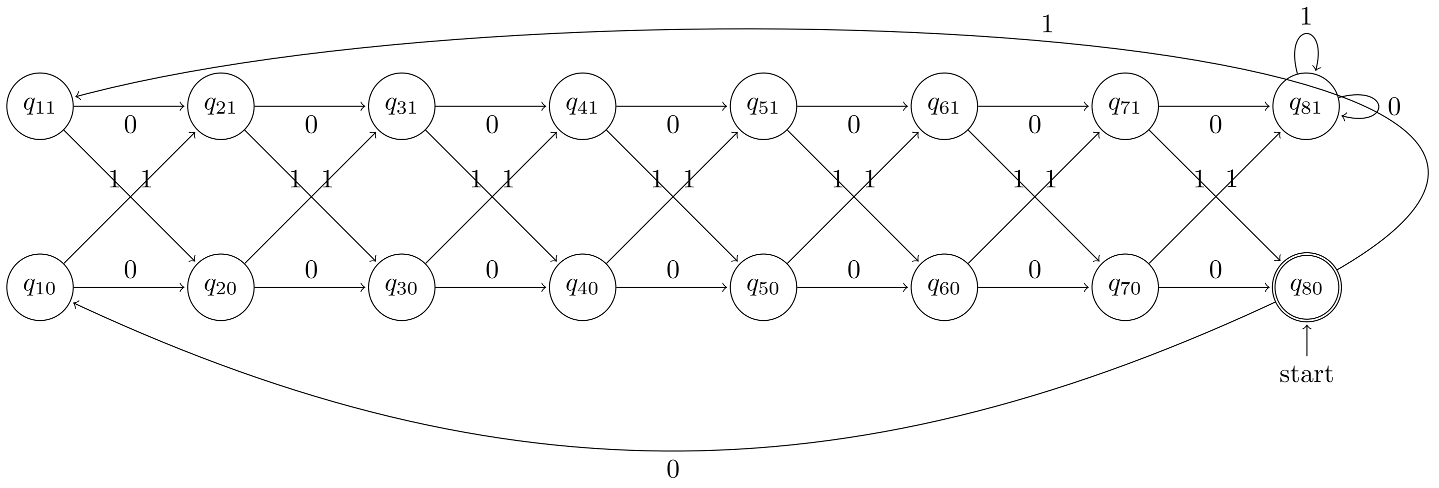
Abstract

1 The model of Latent Dirichlet Allocation

The model of Latent Dirichlet Allocation is a generative probabilistic model for collections of discrete data [1]. Therefore it qualifies to be used for the explorative analysis of text corpora.

1.1 Structure



1.2 Variational Inference

2 Imlementation

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 [2], 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

2.3 Class : LDA

3 Applying LDA

References

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

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

4 Appendix

4.1 Extract subset

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

cd dataset

#Choose how many pages to extract
n=20
#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
```

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
```

```
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.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 => Building Matrix")

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

    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)
        ]

        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", 'rb'))
        self.documentLengths = list(map(len, self.documents))


if __name__ == '__main__':
    dataset = DataSet()
    print(dataset.numOfDocuments())
    print(dataset.dictionarySize())
    print(dataset.matrix.T.dot(dataset.matrix))
```

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


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)
        )

        # 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][wordIndex]

        # 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 step)

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

        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 phi(self):
        """Calculate Parameters of The topic-term multinomial"""

    def theta(self):
        """Calculate Parameters of The document-topic multinomial"""


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