CS145 Howework 6, Naive Bayes and Topic Modeling

Due date: HW6 is due on **11:59 PM PT, Dec. 14 (Monday, Final Week)**. Please submit through GradeScope.

Print Out Your Name and UID

Name: Kevin Li, UID: 405200619

Important Notes about HW6

- HW6, as the last homework, is optional if you choose to use the first 5 homework assignments for homework grading. We will select your highest 5 homework grades to calculate your final homework grade.
- Since HW6 is optional, for the implementaion of Naive Bayes and pLSA, you can choose to implement the provided `.py` and `.py` file by filling in the blocks. Alternatively, you are given the option to implement completely from scratch based on your understanding. Note that some packages with ready-to-use implementation of Naive Bayes and pLSA are not allowed.

Before You Start

You need to first create HW6 conda environment by the given `cs145hw6.yml` file, which provides the name and necessary packages for this tasks. If you have `conda` properly installed, you may create, activate or deactivate by the following commands:

```
conda env create -f cs145hw6.yml
conda activate hw6
conda deactivate
```

OR

```
conda env create --name NAMEOFYOURCHOICE -f cs145hw6.yml
conda activate NAMEOFYOURCHOICE
conda deactivate
```

To view the list of your environments, use the following command:

```
conda env list
```

More useful information about managing environments can be found [here (https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html)](https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html).

You may also quickly review the usage of basic Python and Numpy package, if needed in coding for matrix operations.

In this notebook, you must not delete any code cells in this notebook. If you change any code outside the blocks (such as hyperparameters) that you are allowed to edit (between `START/END YOUR CODE HERE`), you need to highlight these changes. You may add some additional cells to help explain your results and observations.

In [5]:
```
import numpy as np
from numpy import zeros, int8, log
from pylab import random
import pandas as pd
import matplotlib.pyplot as plt
from pylab import rcParams
rcParams['figure.figsize'] = 8,8
import seaborn as sns; sns.set()
import re
import time
import nltk
nltk.download('punkt')
from nltk.tokenize import word_tokenize
from sklearn.metrics import confusion_matrix
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\likev\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Note that `seaborn` in HW6 is only used for ploting classification confusion matrix (in a "heatmap" style). If you encounter installation problem and cannot solve it, you may use alternative plot methods to show your results.

Section 1: Naive Bayes for Text (50 points)

Naive Bayers is one generative model for text classification. In the problem, you are given a document in `dataset` folder. The original data comes from ["20 newsgroups" (http://qwone.com/~jason/20Newsgroups/)](http://qwone.com/~jason/20Newsgroups/). You can use the provided data files to save efforts on preprocessing.

Note: The code and dataset are under the subfolder named `nb`.

In [34]:
```
### Data processing and preparation
# read train/test labels from files
train_label = pd.read_csv('./nb/dataset/train.label',names=['t'])
train_label = train_label['t'].tolist()
test_label = pd.read_csv('./nb/dataset/test.label', names=['t'])
test_label= test_label['t'].tolist()

# read train/test documents from files
train_data = open('./nb/dataset/train.data')
df_train = pd.read_csv(train_data, delimiter=' ', names=['docIdx', 'wordIdx', 'co
test_data = open('./nb/dataset/test.data')
df_test = pd.read_csv(test_data, delimiter=' ', names=['docIdx', 'wordIdx', 'cour

# read vocab
vocab = open('./nb/dataset/vocabulary.txt')
vocab_df = pd.read_csv(vocab, names = ['word'])
vocab_df = vocab_df.reset_index()
vocab_df['index'] = vocab_df['index'].apply(lambda x: x+1)

# add label column to original df_train
docIdx = df_train['docIdx'].values
i = 0
new_label = []
for index in range(len(docIdx)-1):
    new_label.append(train_label[i])
    if docIdx[index] != docIdx[index+1]:
        i += 1
new_label.append(train_label[i])
df_train['classIdx'] = new_label
```

If you have the data prepared properly, the following line of code would return the head of the `df_train` dataframe, which is,

	docIdx	wordIdx	count	classIdx
0	1	1	4	1
1	1	2	2	1
2	1	3	10	1
3	1	4	4	1
4	1	5	2	1

In [35]:
```
# check the head of 'df_train'
print(df_train.head())
```

```
   docIdx  wordIdx  count  classIdx
0       1        1      4         1
1       1        2      2         1
2       1        3     10         1
3       1        4      4         1
4       1        5      2         1
```

Complete the implementation of Naive Bayes model for text classification `nbm.py` . After that, run `nbm_sklearn.py` , which uses `sklearn` to implement naive bayes model for text classification. (Note that the dataset is slightly different loaded in `nbm_sklearn.py` and also you don't need to change anything in `nbm_sklearn.py` and directly run it.)

If the implementation is correct, you can expect the results are generally close on both train set accuracy and test set accuracy.

In [36]:
```
from nb.nbm import NB_model

# model training
nbm = NB_model()
nbm.fit(df_train, train_label, vocab_df)
```

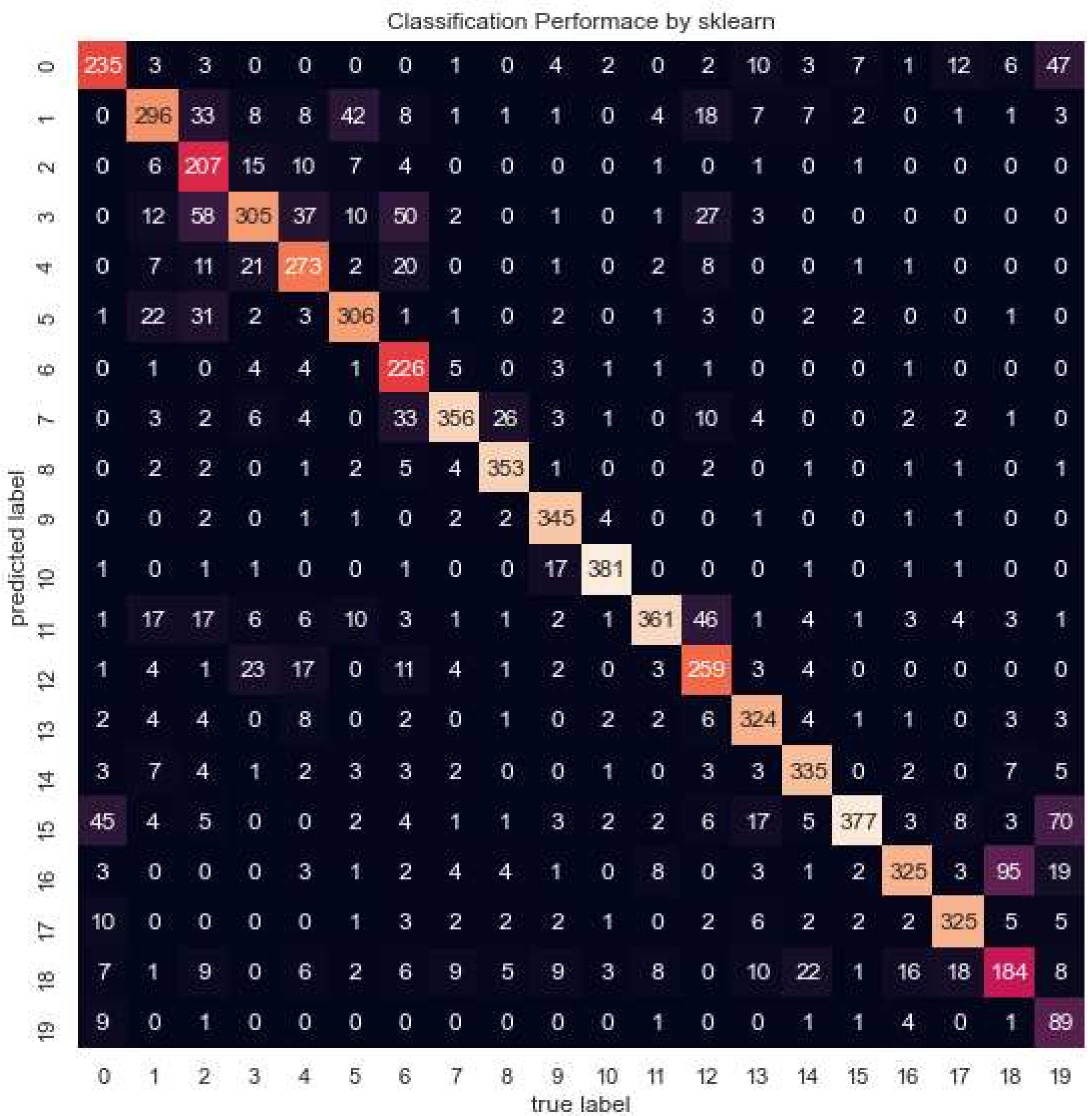
```
Prior Probability of each class:
1: 0.04259472890229834
2: 0.05155736977549028
3: 0.050758718608572219
4: 0.05208980388676901
5: 0.051024935664211554
6: 0.052533498979501284
7: 0.051646108794036735
8: 0.052533498979501284
9: 0.052888455053687104
10: 0.0527109770165942
11: 0.053065933090780002
12: 0.0527109770165942
13: 0.05244475996095483
14: 0.0527109770165942
15: 0.052622237998047744
16: 0.05315467210932647
17: 0.04836276510781791
18: 0.050048806460200055
19: 0.041174904605555506
20: 0.033365870973467035
Training completed!
```

In [11]:
```
# make predictions on train set to validate the model
predict_train_labels = nbm.predict(df_train)
train_acc = (np.array(train_label) == np.array(predict_train_labels)).mean()
print("Accuracy on training data by my implementation: {}".format(train_acc))

# make predictions on test data
predict_test_labels = nbm.predict(df_test)
test_acc = (np.array(test_label) == np.array(predict_test_labels)).mean()
print("Accuracy on training data by my implementation: {}".format(test_acc))
```

```
Accuracy on training data by my implementation: 0.941077291685154
Accuracy on training data by my implementation: 0.7810792804796802
```

In [8]:
```python
# plot classification matrix
mat = confusion_matrix(test_label, predict_test_labels)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False)
plt.title('Classification Performace by sklearn')
plt.xlabel('true label')
plt.ylabel('predicted label')
plt.tight_layout()
plt.savefig('./nb/output/nbm_mine.png')
plt.show()
```



Reminder: Do not forget to run nbm_sklearn.py to compare the results to get the accuracy and confusion matrix by sklearn implementation. You can run `python nbm_sklearn.py` under the folder path of `./hw6/nb/` .

Question & Analysis

0. Please indicate whether you implemented based the given code or from scratch.
1. Report your classification accuracy on train and test documents. Also report your classification confusion matrix. Show one example document that Naive Bayes classifies incorrectly (i.e. fill in the following result table). Attach the output figure `./output/nbm_mine.png` in the jupyter book and briefly explain your observation on the accuracy and confusion matrix.

	Train set accuracy	Test set accuracy
sklearn implementaion		
your implementaion		

2. Show one example document that Naive Bayes classifies incorrectly by filling the following table. Provide your thought on the reason why this document is misclassified. (Note that the topic mapping is available at `train.map` same as `test.map`)

Words (count) in the example document	Predicted label	Truth label
For example, student (4), education (2), ...	Class A	Class B

3. Is Naive Bayes a generative model or discriminative model and why? What is the difference between Naive Bayes classifier and Logistic Regression? What are the pros and cons of Naive Bayes for text classification task?
4. Can you apply Naive Bayes model to identify spam emails from normal ones? Briefly explain your method (you don't need to implementation for this question).

In [76]:
```
lbl_map = pd.read_csv('./nb/dataset/test.map', names=['name', 'lbl'], index_col=1

doc_i, lbl_hat, lbl = next((i + 1, lbl_hat, lbl) for i, (lbl_hat, lbl) in enumera
print(f'Document {doc_i}\nPredicted Label: [{lbl_map.loc[lbl_hat]["name"]}]\nGrou

word_df = df_test[df_test['docIdx'] == doc_i]
word_df['names'] = vocab_df.loc[word_df['wordIdx'] - 1]['word'].to_list()
print(', '.join([f'{name} ({count})' for name, count in word_df[['names', 'count'
```

```
Document 4
Predicted Label: [soc.religion.christian]
Ground Truth Label: [alt.atheism]

atheist (1), of (20), from (1), religion (10), and (25), other (1), are (3),
the (40), in (16), to (19), it (12), like (3), christians (1), on (2), but
(6), with (7), word (2), is (22), people (5), can (3), get (1), for (4), who
(6), go (4), directly (1), bible (1), so (2), one (3), such (1), by (3), or
(4), an (1), which (4), may (1), be (13), humanism (1), secular (3), they
(2), humanist (1), that (12), exists (1), all (4), any (3), well (1), this
(6), example (1), anyone (1), use (1), many (2), thought (2), his (1), at
(7), very (1), he (4), rather (1), than (1), although (2), often (1), had
(1), some (4), god (2), when (1), faith (9), christianity (3), as (6), system
(6), unfortunately (1), whose (1), premise (1), take (1), again (2), under
(1), christian (9), only (2), world (1), down (1), more (1), work (1), has
(3), however (1), probably (2), if (5), you (4), what (4), different (3), sur
e (1), christ (1), seems (1), even (2), university (1), belief (1), also (2),
most (1), case (1), against (2), best (1), without (3), way (2), whether (1),
emphasis (1), dictionary (1), present (1), person (4), philosophy (1), expres
sed (1), over (1), think (2), was (1), reason (2), western (1), values (1), w
ere (2), through (1), those (1), beyond (2), not (9), rationalism (1), mind
(1), become (1), them (1), there (6), will (3), article (1), edu (4), writes
(1), science (2), humans (4), we (5), things (3), put (2), interesting (1), j
ust (3), light (1), much (1), something (1), simply (2), understanding (1), o
k (2), me (1), don (3), anyway (1), cs (3), do (2), no (2), didn (1), good
(4), here (1), admit (2), up (2), definition (1), yes (1), now (1), have (6),
out (2), within (1), should (2), define (3), fail (1), claimed (2), your (1),
claim (1), every (1), implementation (1), seem (1), difference (1), between
(1), still (2), say (1), my (1), personal (2), would (5), reasoning (2), same
(1), because (6), becomes (1), point (2), least (1), notion (1), someone (3),
course (2), too (1), might (1), interpretation (1), wouldn (1), later (1), go
ing (1), result (1), yet (1), kill (1), everyone (1), cause (1), gun (2), alw
ays (3), basically (2), doesn (3), fair (1), perhaps (1), kills (1), really
(3), prison (3), irrelevant (1), due (1), concern (1), inability (1), totally
(1), similarly (1), causes (1), fun (1), presumably (1), free (1), thinking
(3), cannot (3), important (1), realize (1), responsibility (1), maybe (1), w
illing (1), start (1), ask (1), bad (1), problem (1), let (1), past (1), real
(1), open (1), themselves (4), merely (1), consider (1), basic (1), nature
(1), come (1), condemn (1), harm (1), language (1), room (1), need (1), whate
ver (1), please (1), universe (1), change (2), further (1), jesus (1), level
(1), happened (1), apr (1), event (1), experience (1), individual (2), religi
ons (1), believes (1), beliefs (4), whereas (1), unless (2), words (1), leave
(1), neat (1), webster (1), bias (1), himself (1), moment (3), test (1), game
(2), dedicated (1), mass (2), careful (1), minded (1), dogma (7), inherently
(1), adequately (1), philosopher (3), sets (1), adam (1), john (1), cooper
(1), verily (1), laughed (1), weaklings (1), acooper (1), macalstr (1), claws
```

```
(1), prisoner (4), genocide (3), tradition (1), dangerous (1), correspond
(1), extend (1), billions (2), computer (1), nice (1), oriented (1), regards
(1), ahead (2), department (1), divisions (1), semitic (1), rationality (1),
qualify (1), understandable (3), qualities (1), leaves (1), capable (1), anyb
ody (1), granted (1), sadly (1), intuition (1), suicide (1), amazing (1), bet
(1), guarding (1), destroying (1), encourages (4), edt (1), benevolence (1),
waco (1), scorn (1), seemingly (1), bold (1), sects (1), moralities (1), oper
ates (1), evaluated (1), difficulty (1), appalling (1), visible (1), colored
(1), testing (1), toronto (4), losing (1), tests (1), visiting (1), offers
(1), framework (1), skin (1), constrained (1), evaluate (1), poking (1), inve
sted (1), todd (3), chest (1), kelley (2), retaining (1), guise (1), boil
(1), therein (1), bullets (1), nuances (1), hypotheses (1), debated (1), pant
heism (1), arisen (1), differentiated (1), tgk (2), quelled (1), quintessenti
al (1), philanthropy (1), supernaturalists (3)
```

Your Answers

0

My implementation was based on given code

1

	Train set accuracy	Test set accuracy
sklearn implementaion	0.933	0.774
your implementaion	0.941	0.781

2

The table is above. We see that we predicted it was from the christian board, but it was truly from the atheism board. This mistake makes alot of sense - we have a bag of words based model that simply uses word frequencies. This is important for two reasons. First, any religion related boards would be expected to have similar vocabularies used on them. Second, the differentiating factor is likely the sentiment from each board (one may be pro-religion and the other may be anti-religion). However, the bag of words style of model we used does not directly understand sentiment because there is no relationship between when words are said i.e. their context.

3

Naive Bayes is generative because by learning both P(class) and P(words | class) instead of just P(class | words). This allows it to model the joint distribution between the words and classes.

There are several pros of using naive bayes for this task. One major pro is that it is very easy to implement and can still extract very meaningful and complex relationships. Additionally, it generalizes well to smaller test samples/unseen data as long as it has lots of data, which makes it extremely useful.

The cons to using naive bayes are that it simply cannot learn many very important pieces of information from the training data, no matter how much we give it. One clear example is the one we just gave above - it doesn't have any sentiment analysis due to the bag of words style of the model. Mathematically, we also assume that each word is generated independently from each other to simplify the model but in reality, this is untrue and we likely lose some information here as well.

Section 2: Topic Modeling: Probabilistic Latent Semantic Analysis (50 points)

In this section, you will implement Probabilistic Latent Semantic Analysis (pLSA) by EM algorithm. Note: The code and dataset are under the subfolder named `plsa` . You can find two dataset files named `dataset1.txt` and `dataset2.txt` together with a [stopword (https://en.wikipedia.org/wiki/Stop_word)](https://en.wikipedia.org/wiki/Stop_word) list as `stopwords.dic` .

First complete the implementation of pLSA in `plsa.py` . You need to finish the E step, M step and likelihood function. Note that the optimizing process on dataset 2 might take a while.

In [30]:
```
# input file, outpot files and parameters
datasetFilePath = './plsa/dataset/dataset2.txt' # or set as './plsa/dataset/datas
stopwordsFilePath = './plsa/dataset/stopwords.dic'
docTopicDist = './plsa/output/docTopicDistribution.txt'
topicWordDist = './plsa/output/topicWordDistribution.txt'
dictionary = './plsa/output/dictionary.dic'
topicWords = './plsa/output/topics.txt'

K = 4   # number of topic
maxIteration = 20 # maxIteration and threshold control the train process
threshold = 3
topicWordsNum = 20 # parameter for output
```

In [31]:
```
from plsa.plsa import PLSA
from plsa.utils import preprocessing

N, M, word2id, id2word, X = preprocessing(datasetFilePath, stopwordsFilePath) # c
```

In [32]:
```
plsa_model = PLSA()
plsa_model.initialize(N, K, M, word2id, id2word, X)

oldLoglikelihood = 1
newLoglikelihood = 1

for i in range(0, maxIteration):
    plsa_model.EStep() #implement E step
    plsa_model.MStep() #implement M step
    newLoglikelihood = plsa_model.LogLikelihood()
    print("[",time.strftime('%Y-%m-%d %H:%M:%S',time.localtime(time.time())),"]",
          "iteration", str(newLoglikelihood))
    # you should see increasing loglikelihood
    # if(newLoglikelihood - oldLoglikelihood < threshold):
    #     break
    oldLoglikelihood = newLoglikelihood

plsa_model.output(docTopicDist, topicWordDist, dictionary, topicWords, topicWords
```

```
[ 2020-12-14 22:51:32 ] 1 iteration -153235.24336666556
[ 2020-12-14 22:51:39 ] 2 iteration -151625.9443480799
[ 2020-12-14 22:51:46 ] 3 iteration -149731.17563682082
[ 2020-12-14 22:51:53 ] 4 iteration -147804.43324450118
[ 2020-12-14 22:52:00 ] 5 iteration -146137.92941717885
[ 2020-12-14 22:52:06 ] 6 iteration -144851.21438896251
[ 2020-12-14 22:52:13 ] 7 iteration -143916.6692483009
[ 2020-12-14 22:52:20 ] 8 iteration -143265.31326693555
[ 2020-12-14 22:52:26 ] 9 iteration -142810.92022156357
[ 2020-12-14 22:52:33 ] 10 iteration -142470.53495855798
[ 2020-12-14 22:52:40 ] 11 iteration -142197.38511543136
[ 2020-12-14 22:52:47 ] 12 iteration -141968.79801914512
[ 2020-12-14 22:52:53 ] 13 iteration -141776.79306485655
[ 2020-12-14 22:53:00 ] 14 iteration -141624.08864930974
[ 2020-12-14 22:53:07 ] 15 iteration -141502.51512265767
[ 2020-12-14 22:53:16 ] 16 iteration -141400.81789484975
[ 2020-12-14 22:53:26 ] 17 iteration -141317.62208512565
[ 2020-12-14 22:53:34 ] 18 iteration -141251.20311145796
[ 2020-12-14 22:53:41 ] 19 iteration -141197.69014754548
[ 2020-12-14 22:53:49 ] 20 iteration -141155.03431604116
```

In [33]:
```
plsa_model.output(docTopicDist, topicWordDist, dictionary, topicWords, topicWords
```

Question & Analysis

0. Please indicate whether you implemented based the given code or from scratch.
1. Choose different K (number of topics) in `plsa.py`. What is your option for a reasonable K in `dataset1.txt` and `dataset2.txt`? Give your results of 10 words under each topic by filling in the following table (suppose you set $K = 4$).

For dataset 1:

Topic 1	Topic 2	Topic 3	Topic 4

Topic 1	Topic 2	Topic 3	Topic 4
your words	*your words*	*your words*	*your words*

For dataset 2:

Topic 1	Topic 2	Topic 3	Topic 4
your words	*your words*	*your words*	*your words*

2. Are there any similarities between pLSA and GMM model? Briefly explain your thoughts.
3. What are the disadvantages of pLSA? Consider its generalizing ability to new unseen document and its parameter complexity, etc.

Your Answers

0

My implementation was based on given code

1

For dataset 1:

Topic 1	Topic 2	Topic 3	Topic 4
luffy	luffy	island	luffy
devil	pirates	crew	sea
island	haki	manga	grand
fruit	piece	franky	red
pirates	dressrosa	government	blue
""	series	pose	pirates
user	king	straw	burÅ«
crew	treasure	pirates	baroque
fruits	color	set	alabasta
sea	manga	war	piece

For dataset 2:

Topic 1	Topic 2	Topic 3	Topic 4
""	""	""	""
"	"	"	"
soviet	percent	bank	officials
u.s.	rose	percent	california
people	president	bush	people
official	bush	u.s.	dukakis

Topic 1	Topic 2	Topic 3	Topic 4
israel	rate	administration	city
police	government	soviet	union
noriega	economy	trade	barry
government	people	police	president

dataset 1 only has 16 documents, so we should keep k fairly small (>5 probably)

dataset 2 has 100 documents, so depending on what we want to extract/how we want to categorize the topics, we would likely want more (e.g. 10-20)

2

pLSA and GMM have many similarities. One example is that for both, we train them using the EM algorithm. Another (much more important) similarity is that they are both mixture models (in GMM, we assume our distribution is a mixture of gaussian distributions and in pLSA, we assume that the distribution is based on a mixture of multinomial word distributions from topic distributions)

3

One disadvantage of pLSA is that the number of parameters grows linearly with respect to the number of documents (i.e. the training set size). This means that it will struggle with overfitting/will not be as good at generalizing to unseen documents.

Bonus Questions (10 points): LDA

We've learned document and topic modeling techiques. As mentioned in the lecture, most frequently used topic models are pLSA and LDA. [Latent Dirichlet allocation (LDA) (https://ai.stanford.edu/~ang/papers/nips01-lda)](https://ai.stanford.edu/~ang/papers/nips01-lda) proposed by David M. Blei, Andrew Y. Ng, and Michael I. Jordan, posits that each document is generated as a mixture of topics where the continuous-valued mixture proportions are distributed as a latent Dirichlet random variable.

In this question, please read the paper and/or tutorials of LDA and finish the following questions and tasks:

(1) What are the differences between pLSA and LDA? List at least one advantage of LDA over pLSA?

(2) Show a demo of LDA with brief result analysis on any corpus and discuss what real-world applications can be supported by LDA. Note: You do not need to implement LDA algorithms from scratch. You may use multiple packages such as `nltk` , `gensim` , `pyLDAvis` (added on the `cs145hw6.yml`) to help show the demo within couple of lines of code. If you'd like to use other packages, feel free to install them.

Your Answers

1

One primary advantage of LDA is that it the parameter complexity is constant with respect to number of documents, meaning it isn't prone to the overfitting issues that pLSA is.

In [ ]:
```
import nltk
import gensim
```

End of Homework 6 :)

Please printout the Jupyter notebook and relevant code files that you work on and submit only 1 PDF file on GradeScope with page assigned.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

class NB_model():
    def __init__(self):
        self.pi = {} # to store prior probability of each class
        self.Pr_dict = None
        self.num_vocab = None
        self.num_classes = None

    def fit(self, train_data, train_label, vocab, if_use_smooth=True):
        # get prior probabilities
        self.num_vocab = len(vocab['index'].tolist())
        self.get_prior_prob(train_label)
        # ================== YOUR CODE HERE ==========================
        # Calculate probability of each word based on class
        # Hint: Store each probability value in matrix or dict: self.Pr_dict[classID][wordID] or Pr_dict[wordID][classID])
        # Remember that there are possible NaN or 0 in Pr_dict matrix/dict. Use smooth method

        if if_use_smooth:
            Pr_dict = np.ones((self.num_classes, self.num_vocab))
            div = self.num_vocab * np.ones((self.num_classes, 1))
        else:
            Pr_dict = np.zeros((self.num_classes, self.num_vocab))
            div = np.zeros((self.num_classes, 1))

        for i, (docIdx, wordIdx, count, classIdx) in train_data.iterrows():
            Pr_dict[classIdx - 1][wordIdx - 1] += count
            div[classIdx - 1] += count

        self.Pr_dict = Pr_dict / div

        # self.Pr_dict = np.ones((self.num_classes, self.num_vocab)) if if_use_smooth else np.zeros((self.num_classes, self.num_vocab))
        # denom = np.ones(self.num_classes) * self.num_vocab if if_use_smooth else np.zeros(self.num_classes)
        # vals = train_data.values
        # for row in vals:
        #     self.Pr_dict[row[3]-1][row[1]-1] += row[2]
        #     denom[row[3]-1] += row[2]
        # self.Pr_dict = self.Pr_dict / denom[:,None]
        # ============================================================
        print("Training completed!")

    def predict(self, test_data):
        test_dict = test_data.to_dict() # change dataframe to dict
        new_dict = {}
        prediction = []

        for idx in range(len(test_dict['docIdx'])):
            docIdx = test_dict['docIdx'][idx]
            wordIdx = test_dict['wordIdx'][idx]
            count = test_dict['count'][idx]
            try:
                new_dict[docIdx][wordIdx] = count
            except:
                new_dict[test_dict['docIdx'][idx]] = {}
```

```python
                new_dict[docIdx][wordIdx] = count
                ''
        for docIdx in range(1, len(new_dict)+1):
            score_dict = {}
            #Creating a probability row for each class
            for classIdx in range(1,self.num_classes+1):
                score_dict[classIdx] = 0
                # ================== YOUR CODE HERE ==========================
                ### Implement the score_dict for all classes for each document
                ### Remember to use log addtion rather than probability multiplication
                ### Remember to add prior probability, i.e. self.pi
                log_likelihood = np.log(self.pi[classIdx])
                log_likelihood += sum(
                        count * np.log(self.Pr_dict[classIdx-1][wordIdx-1])
                        for wordIdx, count in new_dict[docIdx].items()
                    )

                score_dict[classIdx] = log_likelihood
                # ============================================================
            max_score = max(score_dict, key=score_dict.get)
            prediction.append(max_score)
        return prediction


    def get_prior_prob(self,train_label, verbose=True):
        unique_class = list(set(train_label))
        self.num_classes = len(unique_class)
        total = len(train_label)
        for c in unique_class:
            # ================== YOUR CODE HERE ==========================
            ### calculate prior probability of each class ####
            ### Hint: store prior probability of each class in self.pi
            count = 0
            for label in train_label:
                if c == label:
                    count += 1
            self.pi[c] = count / total
            # ============================================================
        if verbose:
            print("Prior Probability of each class:")
            print("\n".join("{}: {}".format(k, v) for k, v in self.pi.items()))

```

```python
from numpy import zeros, int8, log
from pylab import random
import sys
#import jieba
import nltk
from nltk.tokenize import word_tokenize
import re
import time
import codecs

class PLSA(object):
    def initialize(self, N, K, M, word2id, id2word, X):
        self.word2id, self.id2word, self.X = word2id, id2word, X
        self.N, self.K, self.M = N, K, M
        # theta[i, j] : p(zj|di): 2-D matrix
        self.theta = random([N, K])
        # beta[i, j] : p(wj|zi): 2-D matrix
        self.beta = random([K, M])
        # p[i, j, k] : p(zk|di,wj): 3-D tensor
        self.p = zeros([N, M, K])
        for i in range(0, N):
            normalization = sum(self.theta[i, :])
            for j in range(0, K):
                self.theta[i, j] /= normalization;

        for i in range(0, K):
            normalization = sum(self.beta[i, :])
            for j in range(0, M):
                self.beta[i, j] /= normalization;


    def EStep(self):
        for i in range(0, self.N):
            for j in range(0, self.M):
                ## ================== YOUR CODE HERE ==========================
                ###  for each word in each document, calculate its
                ###  conditional probability belonging to each topic (update p)
                ps = self.beta[:, j] * self.theta[i, :]
                self.p[i, j] = ps / ps.sum()
                # ============================================================

    def MStep(self):
        # update beta
        for k in range(0, self.K):
            # ================== YOUR CODE HERE ==========================
            ###  Implement M step 1: given the conditional distribution
            ###  find the parameters that can maximize the expected likelihood (update beta)
            beta = (self.p[:, :, k] * self.X).sum(axis=0)
            self.beta[k] = beta / beta.sum()
            # ============================================================

        # update theta
        for i in range(0, self.N):
            # ================== YOUR CODE HERE ==========================
            ###  Implement M step 2: given the conditional distribution
            ###  find the parameters that can maximize the expected likelihood (update theta)
            theta = self.X[i] @ self.p[i]
            self.theta[i] = theta / theta.sum()
```

```python
            # ============================================================


    # calculate the log likelihood
    def LogLikelihood(self):
        loglikelihood = 0
        for i in range(0, self.N):
            for j in range(0, self.M):
                # ================== YOUR CODE HERE ==========================
                ###  Calculate likelihood function
                loglikelihood += self.X[i, j] * log(self.theta[i] @ self.beta[:,j])
                # ============================================================
        return loglikelihood

    # output the params of model and top words of topics to files
    def output(self, docTopicDist, topicWordDist, dictionary, topicWords, topicWordsNum):
        # document-topic distribution
        file = codecs.open(docTopicDist,'w','utf-8')
        for i in range(0, self.N):
            tmp = ''
            for j in range(0, self.K):
                tmp += str(self.theta[i, j]) + ' '
            file.write(tmp + '\n')
        file.close()

        # topic-word distribution
        file = codecs.open(topicWordDist,'w','utf-8')
        for i in range(0, self.K):
            tmp = ''
            for j in range(0, self.M):
                tmp += str(self.beta[i, j]) + ' '
            file.write(tmp + '\n')
        file.close()

        # dictionary
        file = codecs.open(dictionary,'w','utf-8')
        for i in range(0, self.M):
            file.write(self.id2word[i] + '\n')
        file.close()

        # top words of each topic
        file = codecs.open(topicWords,'w','utf-8')
        for i in range(0, self.K):
            topicword = []
            ids = self.beta[i, :].argsort()
            for j in ids:
                topicword.insert(0, self.id2word[j])
            tmp = ''
            for word in topicword[0:min(topicWordsNum, len(topicword))]:
                tmp += word + ' '
            file.write(tmp + '\n')
        file.close()
```