FastPlsa: PLSA with Prior

Billy Li
zl20@illinois.edu
Department of Computer Science

December 5, 2021

1 Documentation

FastPlsa is a Python library for performing the PLSA algorithm on a set of documents for classification. It allows the user to use training documents to train priors to improve model accuracy, and features a saving - both manual and auto - and loading feature for robust and convenient execution of the EM algorithm. The most notable aspect of FastPlsa, as its name implies, is its speed: it has been thoroughly optimized to handle large document counts and vocabulary sizes, an example being achieving an average time of 5 seconds per EM algorithm iteration on a subset of the 20newsgroups dataset (number of documents = 1000, vocabulary size = 30000, topics = 20)[1]. FastPlsa is additionally space efficient, storing its models in a compressed manner ensuring that even the largest models can saved in a way friendly to disk space usage.

1.1 Functionality

See demo.py for example code of using FastPlsa.

- **Using FastPlsa:**
  - from FastPlsa import *

  The FastPlsa library depends on the numpy, sklearn and collections libraries.
- **Creating a FastPlsa instance:**
  - FastPlsa(documents_path_test, documents_path_train, labels_path_train)
  - FastPlsa(documents_path)
  - FastPlsa(model_name)

  A FastPlsa instance can be created using the path to the documents to be modeled and optionally the training documents and their labels, or using an existing stored model.
  - Each document in the training/testing file should be a space-delimited string with documents separated by newlines.
  - The label file should have 1 integer (label) on each line, with the line number of each label matching the line number of the corresponding document in the training file.

  - The stored model should be generated from calling the save_model() function of a previous FastPlsa instance.

- **Initializing the FastPlsa instance:**

  - fastPlsa.initialize(random, mixing_weight, number_of_topics)
  - fastPlsa.initialize(random, mixing_weight)

  The initialize function readies the instance for running PLSA by building the corpus and vocabulary from reading the input files, training the prior weights (if training documents are provided), tallying the term-doc matrix, and initializing different weights.

  - If training documents and labels were provided during the creation of the FastPlsa instance, the number of topics are inferred from the number of unique labels in the labels file. Otherwise, it should be explicitly provided during initialization.
  - The mixing weight is the probability that a word is generated from the background model λ_b.
  - The random parameter determines how the weights $(P(z|d), P(w|z))$ are initialized: randomly initialized if true and uniformly initialized if false.

- **Running the EM algorithm:**

  - fastPlsa.plsa(iterations, epsilon)
  - fastPlsa.plsa(iterations, epsilon, save_every_iter, model_name)

  Runs the EM algorithm for the number of provided iterations, and early stops if the likelihood increase between 2 subsequent iterations is less than epsilon.
  If save_every_iter and model_name are provided, the model is automatically saved every save_every_iter iterations using the filename specified by model_name.
  The progress of EM iterations is saved in the instance and persists through subsequent function calls and saving and loading.

- **Checking metadata:**

  - fastPlsa.show_status()

  Displays the metadata of the instance, including the number of topics and documents, vocabulary size, and the number of EM iterations already performed along with the likelihoods of previous EM iterations.

- **Evaluating model accuracy:**

  - fastPlsa.evaluate_model(ground_truth_labels)

  Evaluates the accuracy of the current model using provided ground truth labels for the testing documents. Displays a confusion matrix and the overall accuracy.

  - The ground truth labels should be a 1D numpy array with the indices of the labels matching the line numbers of the testing documents.

2 Project summary

Planned implementations during initial draft of project:

- **Integration with MetaPy (scrapped):** The original project was designed to be integrated into the MetaPy library [2], which is written in C++ and converted into Python code via pybind. However, upon closer inspection, most of the functionality in MetaPy are implementing using optimized multi-threaded code, and implementing PLSA with prior in such a fashion has been deemed unrealistic given deadlines.
- **Extending the PLSA model to include Prior & background model**: This was a pretty straightforward step as the equations are available on lecture slides and notes provided in MP3[**?**]. The input files are read in a fashion similar to SKlearn models with the training documents, labels for the training documents and testing documents read separately. The training documents and labels are used to compute the prior probabilities for each topic and background model.
- **Model saving/loading functionality**: This functionality is implemented by allowing users to store class variables (i.e. $P(w|z)$) using npz, or numpy compressed format. npz compressed variables are around 10% smaller than their counterparts saved by the Python Pickle library. The user can resume an uncompleted E-M algorithm and save time by not having to recompute static variables such as prior probabilities and the background model, as well as preventing unexpected progress loss by specifying intervals for the model to autosave at.

2.1 Efficiency optimization

This is the new focus of the project added after integrating the PLSA with Prior program (now FastPlsa) with MetaPy was deemed unfeasible. A substantial amount of features have been implemented to increase the efficiency of FastPlsa, both in terms of improvements over the original backbone code provided in MP3 and original speedups.

- **Fast normalization:** The normalize function provided during MP3 is not the most efficient way to normalize a matrix by row. The following method speeds up the process by about 50%:
  - normalize(self.document_topic_prob) ## 0.14 seconds
  - self.document_topic_prob /= self.document_topic_prob.sum(axis = 1, keepdims = True) ## 0.07 seconds
- **Efficient counting:** In the backbone code of MP3, the 'vocabulary' variable is specified as a list, with the indices corresponding to different words in the matrices such as the topic-word probability $P(w|z)$ specified by their location in the vocabulary list. This means that for every occurrence of a word, adding the count to the matrices requires calling vocabulary.index(word), which is very inefficient as it is an $O(n)$ operation. FastPlsa instead uses a dictionary to store the vocabulary, with keys being the words and the values being their indices. This allows $O(1)$ lookup and results in an example massive speedup of 30 minutes to 10 seconds when loading the entire 20newsgroups dataset with 200,000 unique words in the vocabulary. Additionally, using the Counter class in the collections library to tally word occurrences for the term doc matrix as opposed to using a for loop to iterate over words in documents further saves approximately 25% time.

- **EM algorithm speedup:** The einsum functionality in the numpy library has been used to both drastically speedup the matrix operations in the EM algorithm and improve the readability of the code. Einsum functions by using the Einstein summation notation as shorthand for matrix multiplication operations, and is significantly faster compared to an equivalent series of chained multiply, dot, and matmul calls.

```
# P(z | d)
for d in range(self.number_of_documents):
    denominator = 0
    for j in range(number_of_topics):
        for w in range(self.vocabulary_size):
            denominator += self.term_doc_matrix[d][w] * (1 - self.background_prob[d][w]) * self.topic_prob[d][j][w]
    for j in range(number_of_topics):
        numerator = 0
        for w in range(self.vocabulary_size):
            numerator += self.term_doc_matrix[d][w] * (1 - self.background_prob[d][w]) * self.topic_prob[d][j][w]
        self.document_topic_prob[d][j] = numerator / denominator

# P(z | d)
numerator = 1 + np.einsum('dw,dw,djw->dj', self.term_doc_matrix, (1 - self.background_prob), self.topic_prob)
self.document_topic_prob = numerator / numerator.sum(axis = 1, keepdims = True)
```

Figure 1: Updating the document topic probability matrix using for loops and the equivalent code using einsum. The former takes 836 seconds to complete while the latter takes a mere 2.7 seconds on the subset of the 20newsgroups dataset (number of documents = 1000, vocabulary size = 30000, topics = 20)

3 Progress summary

- Extending the PLSA model to include Prior & background model: **Done**
- Adding model saving/loading functionality: **Done**
- Reading background/topic models from online sources: **Cancelled**
- Integrating the model with MetaPy: **Cancelled**
- Improving saving/loading functionality using autosave in compressed format: **Done**
- Improving efficiency of PLSA model: **Done**

References

[1] https://github.com/meta-toolkit/metapy

[2] https://scikit-learn.org/0.19/datasets/twenty_newsgroups.html

[3] http://times.cs.uiuc.edu/course/598f16/plsa-note.pdf