CSC 510 Software Engineering: Project 1

Group-P

Monica Metro
North Carolina State University
3021 F Dorner Circle
Raleigh, NC
mgmetro@ncsu.edu

Zachery DeLong
North Carolina State University
2305 Horizon Hike Ct
Raleigh, NC
zpdelong@ncsu.edu

Zhangqi Zha
North Carolina State University
1800 Vienna Wood Dr
Raleigh, NC
zzha@ncsu.edu

Bikram Singh
North Carolina State University
Page Hall, Raleigh, NC
bsingh8@ncsu.edu

ABSTRACT

Choosing a textbook for a class is something that many professors do several times a year, but the vast number of available textbooks for any given subject makes choosing one that both meets the needs of the class and meets the budget constraints of today's students is a challenge. In this paper, we propose two methods for automatically learning topics from books, a set of evaluations for those methods, and a system that will use those methods in a web app that attempts to suggest textbooks to a professor that balance hitting all topics the course requires while still being economical for students.

1. INTRODUCTION

1.1 Background

One of the most basic tasks that a professor must do is identify what textbook to use for a given class. While some fields have textbooks that have become popular and are clearly the best in their subject matter, many fields (especially emerging ones) have no such exemplar and reading through all possible candidates would be too time consuming to be feasible. It would be a complex enough problem if there were not already a massive number of textbooks on sources such as Amazon, but self aware professors have attempted to use textbooks that are less expensive with hopes of allowing lower income students to attend more easily. This admirable intention serves to increase the already considerable time needed to find appropriate books, and there is no obvious heuristic to apply to searches to limit the field.

In the interests of making students lives less expensive, we propose exploring a system which can identify topics in books automatically, and which, given a set of requirements from a user (usually a professor), can suggest options of varying expense and completeness for a given set of topics. To do this, we propose exploring word clusters generated by Doc2Vec, a family of common natural language processing (NLP) algorithms that attempt to cluster similar words, and topics generated by latent dirichlet allocation (LDA), another common NLP algorithm that directly attempts to identify topics in text, to automatically infer the content of a set of textbooks.

We then intend to build a web app that will allow a user to specify a set of topics and which will use the mentioned algorithms to make suggestions while minimizing the cost of said textbooks. The actual web app will be a fairly simple single-page web app developed in Angular4 and using Bootstrap to hasten development of a modern UI.

Initially, the team expects LDA to find more appropriate topics than Doc2Vec simply because it is better purpose built. That said, Doc2Vec uses a sophisticated neural-network, which may behave better on shorter passages such as a table of contents or an index, allowing us to parse less of the book while still getting useful topics.

1.2 Deviations From Initial Plan

Original Plan.

Table 1: Original Schedule

Week started	Task
1-2	Front-end webapp
1-3	NLP CLI
3-4	Suggestion Engine

The team's first attempt at scheduling out the project was focused on using agile methodology. The overall architecture was broken down into components that could be designed and the work was scheduled for two week sprints. This schedule is shown in Table 1. The intent was to develop the NLP algorithms, suggestion engine, and web app in parallel as much as was possible. The original hope was that

PLDI '13 June 16–19, 2013, Seattle, WA, USA

ACM ISBN 123-4567-24-567/08/06. . . $15.00

DOI: 10.475/123_4

the algorithms themselves would be fairly straightforward to work with and that we would be able to apply algorithms for reuse easily to quickly develop what we needed.

Challenges.

It became apparent very quickly that this schedule was not going to work. The team's experience (summarized in table 2 with data mining generally and machine learning specifically was not up to par. While the team had some experience in software engineering, the specific NLP tasks that were required more than anticpated.

Table 2: Developer Skills

Name	Data mining	Programming (years)	NLP
Zach DeLong	1	5	0
Monica Metro	0	0	0
Zhangqi Zha	?	?	0
Bikram	?	?	?

After the project was broken into the above sprints, the team realized they lacked the expertise to split the work up this way.

The Spiral Model.

The solution for our group to our inexperience was to follow a variant of the spiral model. The spiral model is a form of risk-driven development where the most risky parts of a project are identified early and planned into prototypes that can inform the final product. See the breakdown of the prototypes the team chose in section **??**.

This afforded us the ability to have regular check-ins and measure our progress against our goals. The team convened two to three times a week to do pair programming and to check progress against the prototypes we proposed. When combined with regular check-ins with the TAs, we were able to build prototype programs to perform NLP on a set of training data, to identify if we could train a set of models to be applied to textbooks.

1.3 Planning

Prototypes.

The prototypes the team identified for this project are listed in table 3. The team identified the learning algorithms as the most risky part of the project, primarily because of the team's lack of experience with natural language processing algorithms.

As discussed before, these algorithms have succesfully been used to mine topics for some time, but the particular problem of identifying the topics in textbooks for search recommendations is fairly novel. This process requires a data set that matches up well with the desired topics, which will come into play in section **??**.

One other major point of risk is developing the Doc2Vec model. The Gensim API contains a Doc2Vec model but it does not expose parts of the API to make the clusters directly available, which is expanded on in section **??**.

Timeline.

The team completely had to re-evalaute the sechedule to accomodate the prototypes desired in table 3. The team intended to iterate every week, attempting to finish one prototype each week on the way around the spiral model. This often worked, and prototypes 1, 2, and 3 were finished in the first three weeks of the project. Prototypes 3 and 4 were develoed in parallel over the last several weeks of the project and it was followed immedaitely by implementing the evaluation plan in section **??**.

2. LITERATURE REVIEW

We propose two methods for analyzing and storing topics from books automatically: LDA and Doc2vec.

2.1 LDA

In this section, we describe the Latent Dirichlet allocation(LDA) method for storing topics against books.

LDA uses a Bayesian clustering approach to find topics relevant to the book. [3] The input to the method is a book (or later, a series of books) which consists of collection of words.

This method assumes that each book (or books) consists of topics and that each topic has several key words associated with it. The number of topics associated with a book can be variable, and can be changed for better optimization. In this association, the words comprising the text are thrown into a "bag-of-words" model such that grammer and word order are disregarded, but multiplicity is measurable. The calculated topics found from this bag-of-words are "latent" variables, that is they are not directly measurable but indirectly observed.

The performance of the method depends on the initial assumptions made. For example, some have assumed that books are random mixes of topics [2]. The number of topics can initally be chosen as constant for a given book, or as a function of a chosen Poisson distribution. [2] Topic distribution in the book is assumed to be a sparse Dirichlet function.

A sparse Dirichlet function implies that we assume that a topic will be discussed only in a small breadth of pages in the book, and also that words relating to this topic will predominantly figure in these pages.

The algorithm looks to identify unique words to identify as topic names. Words like "the" and "a" are common and will occur with equal probability throughout the text. However, in say one chapter of the book some words may occur much more frequently in that particular chapter than the rest of the book, which means that the probability of these words in that chapter is more, which then the algorithm can use to choose a topic name and also a list of words related to that topic.

2.2 Doc2Vec

2.2.1 Introduction

Doc2Vec is an implementation of the Paragraph Vector (PV) introduced by Mikolov and Le in 2014 - an unsupervised algorithm that generates vector representations of text inspired from research of vector representations of words via neural networks. [5] By concatenationg or averaging vector represenations of words with other word vectors, the resulting vector can be used to predict future text. This technique offers one distinct advantage to bag-of-words models like LDA: the ability to record semantics due to the mapping of the word vectors into a vector space that allocates words of similar meaning together. For example, the words "powerful, strong and Paris" are equally distant when considered by a bag-of-words model, however "powerful" and "strong" are semantically close and should therefore be less distant from one another. [5]

PV is "unsupervised", which means that it predicts words in a paragraph and then uses these predictions to attempt to form structures of related words without any knowing of accuracy of the structure(s). The PV utilizes fixed length feature vectors learned from text sources of variable length.

2.2.2 *Models*

Mikolov offers two models of PV that are based on the implementation of Word2Vec: Distributed Memory Model (PV-DM) and Distributed Bag of Words (PV-DBOW).

PV-DM is similar to the Continuous Bag of Words (CBOW) model in Word2Vec such that is predicts the next word based on the given context. Paragraph vectors are concatenated with word vectors from that paragraph. Each word vector is representing a word from the given context and a new vector is generated by concatenating all of those word vectors together to predict other words. These word vectors are able to remember context and semantics. The paragraph vector allows the algorithm to remember the topic or 'label' of the paragraph, whether it is a sentence or a long document.

PV-DBOW follows the skip-gram model of Word2Vec. In this model, instead of concatenating the paragraph vector with the word vectors formed in close proximity in order to predict the next word, random text from the paragraph is selected and a random word is selected from that text. Because this model does not save the word vectors and therefore does not retain as much information about the context and semantics of the text, it requires less storage.

2.2.3 *Application*

Mikolov and Le recommend using a combination of PV-DM and PV-DBOW for consistently accurate results. [5]

2.3 Related Work

Doc2Vec was recently tested as the algorithm behind a recommender system with the goal of recommending unseen Twitter messages (tweets) relative to a user's typical activity and interests. A graph based link prediction method was used to infer which unseen tweets the users would like by "predicting the presence or absence of edges between nodes" on the graph. [1] K values were tested in intervals divisible by 5 ranging from 5 to 35. The system preformed the best when K=30.

An LDA approach of a tagging system was implemented to improve tag recommendations. [4] Tagging systems are often used for organizaing a user or organization's data. When this data is shared, tagging can be used to find or search for relative content.This system is similar to our proposed book recommendation system that uses subject topics to search for relative material. The initial tag data set was comprised of a large sample creating a diverse tag set. This set was used to elicit latent tags of sources that did not have that many tags to describe the document. Often, the tags recommended were more specifc. By increasing the number of tags and increasing specific tags, their approach contributed to the usefulness of searching for new content.

3. ARCHITECTURE

3.1 System Design

Overall System.

We propose building a layered system where each piece of the funcionality of the overall application is built into one layer of the overall design.

This design gives us the ability to build the riskiest, most difficult pieces of functionality early, while slowly building up to an application taht is usable. It also allows us to parallel path much of the development of the overall system. One person can easily be working on one layer of the system without impacting other layers, as long as interfaces between layers are well designed and documented.

The other primary advantage of the layered model detailed in the following sections is that it makes validation possible at each layer of the model. As we will get into in the following sections, the overall application will need to be validated as it is built, for the accuracy of its NLP algorithms, the usefullness of its suggestions, and the user-friendlyness of it's UI

See figure 3.1 for a detailed breakdown of the overall system design. The arrows in this diagram represent the flow of information from one layer to the next. Notice how no layer communicates with anything past its surrounding layers.

Textbook Library.

NLP Algorithm.

The "lowest" level of the stack we have designed is the suggestion algorithms. This layer has the responsibility of identifying the topis found in textbooks using trained models. Because all modern topic modeling algorithms are extremely slow to run, analysis can not be done in real-time. To work around that, we propose buildign a command-line application that could be scheduled on a server to run on some kind of regular interval and set against some external textbook parsing API. See section 3.2 for a more detailed breakdown than this overview.

Database.

The database sits above the NLP algorithms, but below the rest of the stack. Its purpose is to hold the topics, textbooks, and author information mined from our external API. To house this information, we have assembled a database using the Python library Peewee as an ORM. When run, Peewee will build out the tables listed in figure 3.1. The app proposed will use the database as a repository from which to make suggestions rather than natively running the NLP algorithms, which should help to improve the performance of the overall app.

Suggestion Algorithm.

The suggestion layer is the last layer of the application before the interface itself. To make suggestions, the team intends to build out a REST API built on top of the database previously discussed. The API should accept a list of topics to search for along with two weights: one for the wieght of topic coverage and one for weight of cost.

The weights serve as tuning parameters and may or may not be displayed to the end-users. The weights are them multiplied by their respective parameters and summed to come to a final priority number, which can then be consid-

Figure 1: Visualization of layered architecture

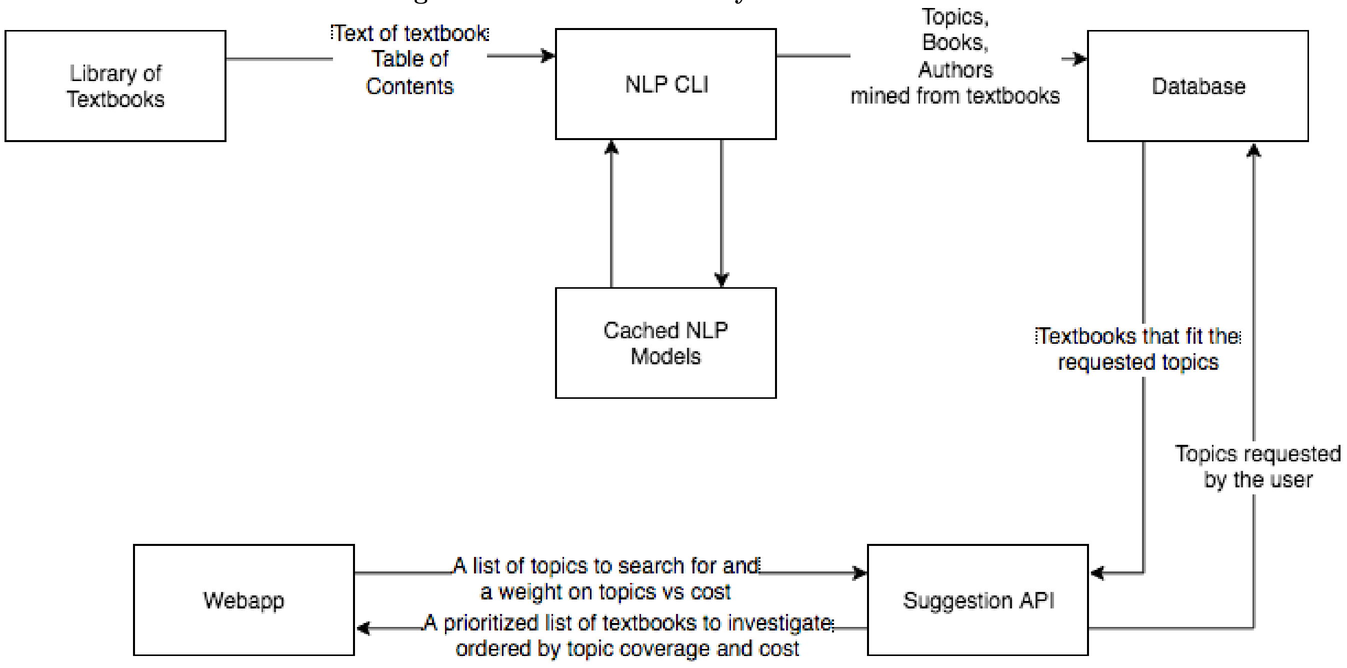
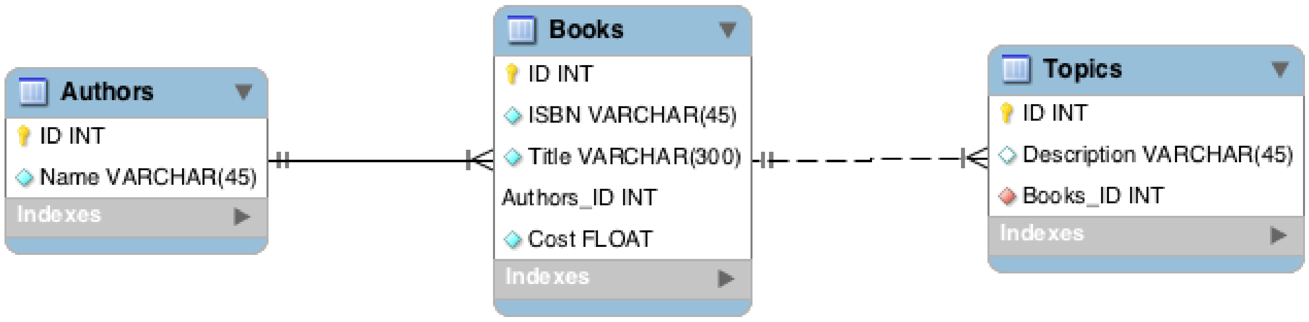

Library of Textbooks
Text of textbook Table of Contents
NLP CLI
Topics, Books, Authors mined from textbooks
Database
Cached NLP Models
Textbooks that fit the requested topics
Topics requested by the user
Webapp
A list of topics to search for and a weight on topics vs cost
A prioritized list of textbooks to investigate ordered by topic coverage and cost
Suggestion API

Figure 2: ERD of Proposed Database



ered a measure of the "fitness" of a particular textbook for use, where larger numbers are more fit. The fitness numbers should also be normalized to a scale between 0 and 1, so that the "fitness" of different searches could be compared in a meaningful way.

The proposed formula for calculating the priority of a suggestion is detailed in equation 1.

$$Priority_i = a \times num\_topics_i + (b \times max(cost) - cost_i) \tag{1}$$

Web App.
The final componenet in this proposed architecture is the user interface. For the purpose of this project, the team proposes building a simple webapp using AngularJS. In this design, the actual web app needs to do very little; it is merely acting as an interface to the previously mentioned API. It does not need to directly connect to a database or to perform complex calculations, it simply needs to provide a simple, easy way to input a set of search topics and to display the results of the REST API to the users.

AngularJS is well positioned to be useful in this situation. It provides convenient APIs for the display and manipulation of complex interfaces, without providing significant bloat to the application stack duplicating the features of the previous components. It is also structured as a model-view-controller system, so our manipulation of the REST api can be well and cleanly separated from the presentation, allowing us to develop other applications easily in the future.

3.2 CLI Design

The NLP algorithms are the focus of most of the rest of this paper. To make it easier to test and evaluate these algorithms, the team designed a CLI using the Python package Click. Once the CLI was up and running, the learning algorithms were implemented using the package Gensim. Wrapper classes around the Gensim models were built using a variant of the strategy pattern, so that our implementations could easily be swapped back and forth during testing.

3.3 Use cases

The team envisions fairly few use cases for the proposed system, as the system's scope is fairly well defined. The system recommends textbooks to users based on the textbook cost and the topics covered in that textbook. With that very limited scope in mind, the webapp needs to support three use-cases documented in table 4

4. PROGRESS

5. VALIDATION

6. EVALUATION

In this section, we present the evaluations that been performed on our solustions. This including two parts: 1) methods that used to evaluate our core topic learning models LDA and Doc2Vec and 2) methods used to evaluate the whole solution as the final product.

Evaluating Learing Models.

The performance measure is a typical way to evaluate machine learning models. It is the measurement that will make of the predictions made by a trained model on the test dataset. Performance measures are typically specialized to the class of problem, for example in this paper, our problem is a NLP topic modeling, we will use likelihood score and perplexity to evaluate our LDA model.
Based on how to split the data into training and testing, evaluate machine learning models usually involves hould out, cross valiation(CV) and leave-one-out.

6.1 Evaluating LDA

6.2 Evaluating Doc2Vec

While LDA produces keywords predicted to be the main topics of the document, Doc2Vec produces vectors. To convert these vectors in potential important topics, first the accuracy of the model must be established to ensure that it can distinguish between one topic and another. Two tests were used to evalutate the accuracy of the three Doc2Vec models, Distributed-Memory by taking the mean of the vectors (DM-M), Distributed-Memory by concatenating (DM-C), and Distributed Bag of Words (DBOW): vector prediction and information retrieval.

Doc2Vec was tested on parsed stackoverflow data. Folders of different tags were created such that no document had the other tags that were being tested listed. For exaple, a document tagged 'python' would not also have 'javascript' as a tag. This was done to decrease topic blending and increase the semantic importance of the tags. Tags included python, javascript, database, c#, .net, and java.

6.2.1 *Vector Prediction Accuracy*

6.2.2 *Information Retrieval*

One way to verify that our implementation of Doc2Vec is correctly mapping word vectors to other semantically similar word vectors is to test the trained model on an information retrieval task . [5] The data collected from stackoverflow is tagged with different topics that pertain to the content. The tags can then be used to organize the content via topic. The Doc2Vec model should report two documents that pertain to the same topic as less distant to each other than a third document that focuses on a different topic.

Three different datasets were used for this test largely based off the results of the vector prediction analysis testing: a set trained on just Python, a set trained on Python and Database, and a set trained on Python and Javascript. The Python and Javascript training data contained 25,000 randomly selected documents from each tag. Another 15,000 documetns for each tag that were not used for training were selected to be tested. Out of these 15,000 documents, two of the javascript documents were shown with one python document to determine if the model could accuratlely determine a semantic difference between the two tags allowing the model to be used for topic prediction like LDA. The same was one with the python and database set, except only 2,000 database documents were tested against 4,000 python documents shown two at a time. The rest of the python and database documents were used for training. The number database documents was far less than both python and javascript, but the database dataset scored high on the vector prediction analysis indicationg a possible semantic advantage over python and javascript. The set trained on just python was tested the same way as the other python and database set.

Mikolov et. all Used two different sources for their information retrieval tests ——CITE—-. One was sentences taken from Rotten Tomatoes reviews that were labeled by MTurk workers as positive or negative. The models preformed well with being able to tell if a sentence was negative or positive, but not with five fine-tuned ratiings - if it was very positive vs. slightly positive, etc. The paragraph vector percent error for the fine-tuned ratings was a high 51.3%. The other test involved taking 1 million popular search engine queries and creating three paragraphs several sentences long from the 10 most popular results for each query. Two of the paragraphs would be from the same source while the third would be from a different source. Their implementation of the paragraph vector was able to correctly idenity the paragraphs from the same source as closer in distance with the vector space with only a 7.42% error.

Similar to the fine-tuned Rotten Tomatotes experiment, attemps to implement Gensim's Doc2Vec were met with high percent errors with the parsed stackoverflow data. The error rate was determined by two measurements: the Eucildean distance and cosine simularity. Eucildean distance takes two vectors and calculates the straight-line distance between them. This measurement was used initially with poor results. The number of epochs was increased from 20 to 50 iterations in an attempt to better performance, but the error consistently became worse. The only variable that increased performance was making the number of feature vectors smaller. Upon further research, it was discovered that within a vector space, Eucildean distance is not always an accurate measurement because two vectors that are close to each other will be classified as similar even if they are going in opposite directions, thus actually being opposites in semantics. Similar semantically meaning words can also be deemed far apart if the texts vary in length greatly resulting in a high word counts creating a larger magnitude vector. Cosine simularity, the measurement Gensim uses to compare inferred vectors to the trained vectors within the models, was adopted to solve this problem.

Cosine simularity is usually used to compare vectors by the size of the cosine angle between the vector and the origin. Comparing to the origin captures the semantical direction and accounts for the magnitue problem. However, in this test the Eucildean distance consistently performed better than the cosine simularty. —-FIGURES— Raising epochs from 20 to 50 only seemed to affect the DM-C model, decreasing performance. The two datasets that were tested with the python and database tags performed better than python and javascript at Eucildean and cosine distance even though the python and javascript dataset was implemented with less epochs, which seemed to make the DM-C model worse.

6.2.3 *Conclusions*

A high error rate when trying to distinguish between topics suggests that there may be no actual semantic relationship between the stackoverflow data. Part of the problem might be the topics themselves. Python and javascript, for example, are both computer programming languages. Although they have different uses and purposes, the concepts surrounding them are more similar than say a programming language and more distance subject such as biology. To increase the semantic relationship, the stackoverflow data could be parsed more effectively. The data consisted of the 'posts' from stackoverflow. These posts did not contain the post titles or the comments. Both the titles and comment so often contain the most helpful information relevant to the topic at hand.

Content with more specific information that is better suited for parsing could also be adopted instead. For example, typo-less information or cleaned up code from scholarly resources might allow for more efficient tokenization of relevant words or symbols like parathensis and brackets which are more prevalent in some programming languages than others. It is possible, however, that because the data from stack overflow comes from an untold amount of users, a semantic relationship may not be able to be determined.

!!!Doc2Vec was not found to be successfull at predicting topics because of a lack of semantic relationships between datasets. Could be used to find already trained on and labeled data (VPA results) —— If the model was accurate, could be used on pre-labeled trained data to suggest the labels from the most similar document to the inferred vector of the document being compared –> Put in last papaer conclusion !!!!

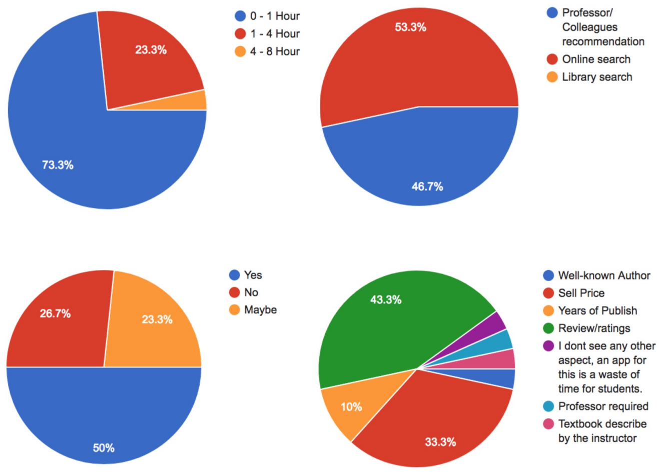
6.3 Evaluating Application

Pre-Surveying.

To evaluate our project idea, we conducted a pre-surveying to see what the problem we are facing and how people reacting about our proposed solution. We asked our participants how long they will spend on find the right textbook, what aspect do they think is most important when choosing a textbook (except the content, our learning model will do the best of this part). See Table 5 for a breakdown of the questions to ask.

Among the 29 responses, we found that 73.3% responded as they interested to our application to recommend the textbooks. The answer to most important aspect during the textbook chosen, review/ratings and sell price were the highest concern. These preliminary results give us the guidance to design the application and to improve the core learning models. See Figure 3 for detailed results.

Figure 3: Result of preSurvey



Experiments and Post-survey.

After we build the application with the command line interface and core learning models, we ran the experiments with different users in a foucs group to evaluate the solution.

7. CONCLUSION

8. REFERENCES

[1] M. Z. Amiri and A. Shobi. A link prediction strategy for personalized tweet recommendation through doc2vec approach. *Research in Economics and Management*, 2(4):14, Jul 13, 2017. Doc2vec works better than other âĂŸapproachesâĂŹ with tweets.

[2] A. Y. N. David M. Blei and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2000.

[3] M. S. Jonathan K Pritchard and P. Donnellyu. Inference of population structure using multilocus genotype data. *Genetics*, 155(2):945–959, June 1, 2000.

[4] R. Krestel, P. Fankhauser, and W. Nejdl. Latent dirichlet allocation for tag recommendation. RecSys '09, pages 61–68. ACM, Oct 23, 2009.

[5] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. May 16, 2014. Paragraph vector (doc2vec).

Table 3: Prototypes

1	• Implement CLI to make testing of models easier • Implement LDA algorithm using Gensim on static text files.	3	+
2	• Prototype 1 • Parse Stack Overflow data to text files for input to prototype 1 • Save/Load model so multiple experiments can be run more quickly • Adapt Prototype 1 to operate on Stack Overflow data.	4	+
3	• Prototype 2 • Prototype database models/access code. • Prototype similarity measures to be used with learning algorithms	4	+
4	• Implement Doc2Vec • Implement Doc2Vec with stack overflow data	5	+
5	• Implement code to parse PDFs of textbooks • Implement connection to library (either REST API or a folder of PDFs • Apply trained models to textbooks • Save modeled topics in the database	4	-
6	• Write algorithm to parse saved topics • Write REST API to serve above algorithm	2	-
7	• Build web-app front end to the above suggestion service	2	-

Table 4: Use Cases

user	task
Professor	Search for textbooks with a given set of topics
Professor	Search for textbooks with a given set of topics using an increased weight on providing an inexpensive textbook
Professor	Search for textbooks with a given set of topics using an increased weight on matching all topics requested

Table 5: Questions in Pre-survey

Questions
1. Are you a student or professor?
2. How much time do you spend on choosing a textbook?
3. How do you choose a textbook?
4. What aspect do you think is most important when choosing a textbook (except the content)?
5. Would you consider an app that will give you recommended textbooks base on your course syllabus and your preference?