Brandon Cuadrado, 109237297

Pranavi Venkata Changamma Meda, 111492602

Zenab Bhinderwala, 109897840

Automatically Building Book Indices: Progress Report

Introduction

An index is an alphabetical listing of words or phrases (usually key words) with references to the places/page numbers where they occur. The goal of this project is to develop an automatic index builder; which takes a LaTeX document and the desired index size as input and outputs an index in a new LaTeX document. The application will use a model learned from existing LaTeX indices to predict the appropriate content for the generated index. The automatic index builder will be a command line application developed using Python 2.7.

Reading LaTeX Files

In the Project Proposal, we suggested TexSoup as the tool to parse LaTeX data from existing LaTeX documents. However, our current parser for reading LaTeX files uses a tool known as pylatexenc. Pylatexenc is capable of outputting the visible text of a compiled LaTeX file, among other functions. Pylatexenc can perform this operation using its LatexNodes2Text class. The LatexWalker class can be used to obtain more detailed information about the document.

```
    # Get only the text from the file
    latextext = file.read().encode('utf-8')
    text = LatexNodes2Text().latex_to_text(latextext)
    lw = LatexWalker(latextext)
```

The 17 LaTeX documents obtained in preparation for the Project Proposal are used to retrieve data on its words and phrases that are eligible for an index. These same files are used to obtain their current indexes, which will be used to train and test our models as they progress.

Determining Index Terms

Index terms are categorized as single words or phrases that contribute to the context of the document. The baseline model implements a scoring function using the data collected from the LaTeX file, organized by word or phrase. The success of this model will be determined using precision and recall of the results.

Obtaining Terms

Unigrams

The potential index terms are obtained by splitting the text variable as obtained above using LatexNodes2Text. Unigrams refer to the single words obtained from the LaTeX file. The text variable is split by various possible characters using the following splitText function.

```
# split text using an array of delimiters and check validity
def splitText(text):
    # delimiters - only for characters that separate words
    # conjoining characters not included
    splitChars = {'\.', ' ', '\,', ':', ';', '\n', '\*' \
        '\|', '\?', '\(', '\)', '\^', '_'}

    splitString = ""
    for char in splitChars:
        splitString += char + '|'

    # spilt string
    words = re.split(splitString, text)

    return words
```

Bigrams

In addition to single words, phrases of two words are included in the dataset. As our model progresses, we intend on including n-long phrases in the dataset. Currently, only phrases consisting of two words are added. NLTK, the Natural Language Toolkit, can extract two word phrases using its bigram function.

Data Processing

Data Cleaning

Following this operation, the checkWords function is used to filter invalid words from the list of unigrams and bigrams. This function is applied to all elements of the words list using the internal filter function. First, checkWords confirms that the word can be converted to a string. Then, it is confirmed that the word is not a stopword or one of the 500 most common English words. The word is then determined to be neither a single character nor a number. Finally, the word is matched against a regular expression such that the string contains at least one letter or number.

```
# filter invalid words from list of words
def checkWords(word):
    try:
        word = str(word)
    except:
        return False

    if word.lower() in nltk.corpus.stopwords.words('english'):
        return False

    if word.lower() in common500:
        return False

    if word.isdigit():
        return False

    if len(word) < 2:
        return False

    pattern = re.compile("([A-Za-z0-9]+)")
    return pattern.match(word)
```

N-Gram Score

Each term is given an n-gram score based on the number of words that appear in the term. Unigrams receive a value of 2, whereas bigrams receive a value of 10, based on their weight in the scoring function. For future models, the score will correspond with the number of words in the phrase. This will allow the weight of this attribute to vary between models.

Part of Speech Tagging

Each term is classified using a Part of Speech tag, as defined by NLTK. These tags represent parts of speech such as “Noun”, “Adjective”, and more specific subdivisions such as “Proper Noun”. Each tag has a two or three letter code associated with it. This can be done for words and for entire phrases using the pos_tag function offered by NLTK.

Calculating Frequency

The term frequency is defined by the number of times it appears within the text. Due to words potentially having different POS Tags based on context, the frequency is calculated based on the unique word, n-gram index, and pos tag pair. This can be achieved by grouping a data frame by the paired values and calculating the size of each group.

The following code snippet details how frequency is calculated for each term:

```
# Group words by their word, n-gram index, and pos, then count number of occurrences
df = df.groupby(['word','n_gram_score','pos']).size().reset_index(name="frequency")
```

Term Frequency * Inverse Document Frequency

This feature compares frequency of phrases used within an individual document and the frequency of that phrase in general use among multiple documents. This is also a metric of informativeness, which is defined below. Term frequency is the frequency of the term in the document, whereas document frequency is the number of documents containing the phrase.

Informativeness

Investigations in Unsupervised Back-of-the-Book Indexing[1] defines informativeness as the degree to which a key phrase represents the document at hand. It correlates with the amount of information that a specific phrase conveys to the reader. This involves the following procedure, which calculates Kullback-Liebler divergence for each term.

```
# Calculate informativeness for all terms
inf_list = []
for i in term_freq:
    inf = i * math.log(i * idf , 2)
    inf_list.append(inf)
df_final['inf'] = inf_list
```

Parser Interface

The Parser Python 2.7 application uses a command line interface. The '-f' flag allows the user to input a single LaTeX file as the input and retrieve a single output file as described by the '-o' input flag. Using '-d' provides the same functionality, but for an entire directory of LaTeX files. This allows for successive parsing of multiple LaTeX documents using a single call of the Parser.py script.

[1] http://web.eecs.umich.edu/~mihalcea/papers/csomai.flairs07.pdf

Existing Indices

Parsing Index Data

The indices of existing LaTeX files can be parsed using the CreatingIndex.py script. This program runs through all the LaTeX files within a directory and parses the file for all /index tags. It also considers user-defined tags which include the /index command when performing this action. These indexes and data associated with them are outputted to an output csv file for comparison against the general data parsed from the document.

Index Analysis

Within Jupyter, the data obtained from indices is cleaned to include eligible words. Currently, mathematical symbols, which appear often in indices papers regarding mathematical concepts, are beyond the scope of our prediction model. As such, the indices are cleaned to include eligible terms, then tagged as Parts of Speech and other information gained from the data. The data obtained from existing indices is used to determine the weight of each attribute when scoring terms.

The following graph represents the distribution of Parts of Speech among the observed indices in existing documents with an index.

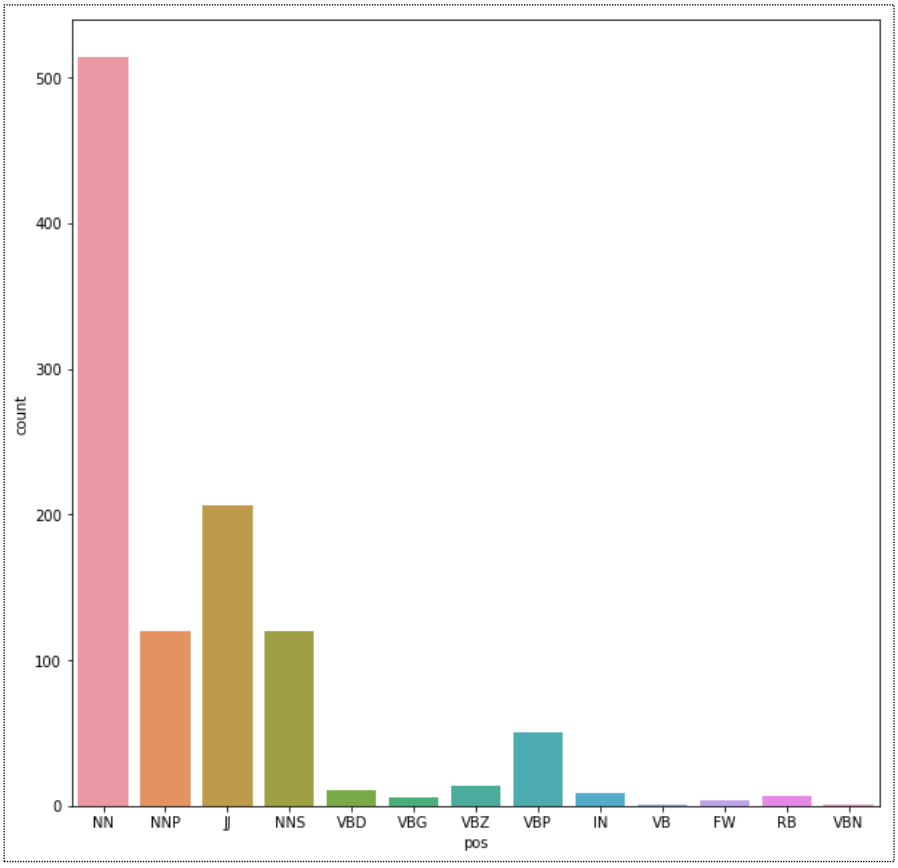

Figure 1

Baseline Model

Scoring Function

This scoring function uses each term's pos_tag, term frequency, and informativeness to evaluate its significance. From Figure 1, we found that nouns and adjectives populate indices more than verbs and other parts of speech. As a result, a comparative weighted score has been given to terms based on their parts of speech to reflect this finding.

Bigrams, trigrams and n-grams, containing up to 5 words, comprise most of the indices compared to unigrams. Due to this, a weighted score is assigned to all terms based upon the degree of n-gram. Unigrams are given lower importance compared to bigrams and trigrams.

As of now we have considered only unigrams and bigrams. The formula is as follows:

S_t = ((pos_rank * n_gram_score) + inf) * tf_idf)

Where; Pos_rank = rank based on pos_tag

N_gram_score = score based on degree of n-gram

Inf = informativeness

Tf_idf = term_freq * inverse document frequency

The top 50-100 words based on the score are placed in the indices. Based on parts of speech, the following bar plot represents the frequency of each parts of speech within an index. The plot compares the frequency observed for the actual index against the predicted index.

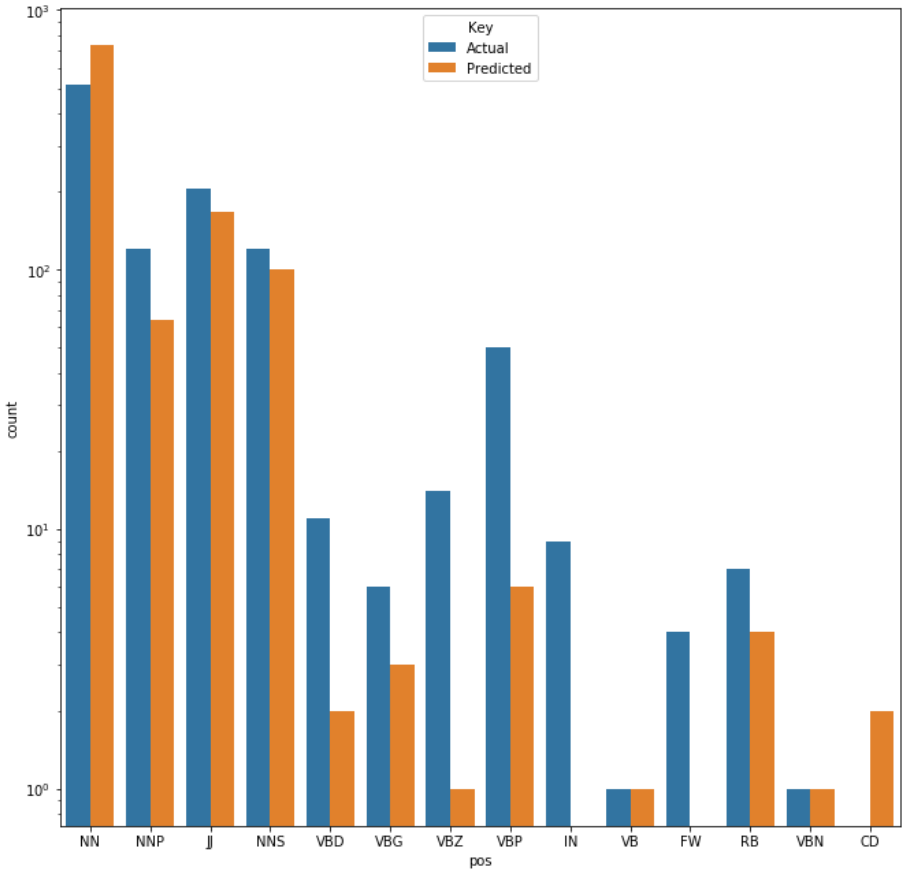

Figure 2

Evaluation

Precision and Recall are useful measures to evaluate the quality and success of a prediction model. They measure how well an information retrieval system retrieves the relevant documents and features requested by a user. The Precision represents the number of terms retrieved that are relevant, divided by the total number of terms that are retrieved. The Recall represents the number of terms retrieved that are relevant, divided by the relevant terms in the database. In the following equations, “tp” represents True Positives, “fp” represents False Positives, and “fn” represents False Negatives.

$$Precision = \frac{tp}{tp + fp} \qquad Recall = \frac{tp}{tp + fn}$$

Currently we have considered only unigrams and bigrams as a part of indices. With the baseline model (scoring function) to predict the indices, we got a precision of 0.4838 and recall value 0.4921.

Final Index Creation

Once the index terms are selected by our model, a python script will loop through a LaTeX file and add an index tag to the chosen words. A copy of the original LaTeX file will be created with the added package of makeidx, using the command "\makeindex". To print the index, the command "\printindex" will be added to the end of the document to append the automated index.

```
import subprocess

# scribe.tex represents LaTeX file with added \index commands
file = open('scribe.tex', 'r')
# create scribe.idx - represents index with \indexentry tag
subprocess.call('pdflatex scribe.tex', shell = True)
# create scribe.ind - LaTeX representation of index
subprocess.call('makeindex scribe.idx', shell = True)
# create pdf for index entries
subprocess.call('pdflatex scribe.tex', shell = True)
```

Future Work

Now that there is a baseline model, further models will be generated to better predict indices. Such models include classification models, such Naïve-Bayes Classification, and regression models. Additionally, the Index Creation Python script will be fully developed based on the code described above. As this work is done, we will also refine and build upon the data obtained from the LaTeX document in the Parser.py script.