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 first 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
```

Following this operation, the checkWords function is used to filter invalid words from the list of strings. 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)
```

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. These are then filtered using the checkWord function on both elements of the bigram.

Data Processing

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 dataframe by the paired values and calculating the size of each group.

```
# 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")
```

Document Frequency

The Document Frequency is defined as the frequency of the word in the document, displayed as a decimal value. This represents the fraction of the frequency over the total eligible words.

Informativeness

Investigations in Unsupervised Back-of-the-Book Indexing[1] defines informativeness as the degree to which a keyphrase 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.

```
# 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.

Existing 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.

Scoring Function

Evaluation

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