Document Clustering

Anindya Dutta and Aarushi Goel

November 14 2017

Contents

1 High-level summary

1.1 Work done so far

In our last report, we had pickled our list of books and saved it for further analysis and unsupervised learning. In this report, we start by describing the basic architecture of the project that has been developed over the previous few reports. We describe how all the scripts are related, and how data flows from one script to the next. We have uploaded the project on GitHub at this link.

1.2 Unsupervised learning on data

We have referred various sources before deciding on using k=means clustering as the first step towards classifying the data. Our work is inspired by Brandon Rose's project on clustering movie reviews from IMDB. Our data therefore goes through the following steps, which we will describe in detail in this report:

- tokenizing and stemming the cleaned up text

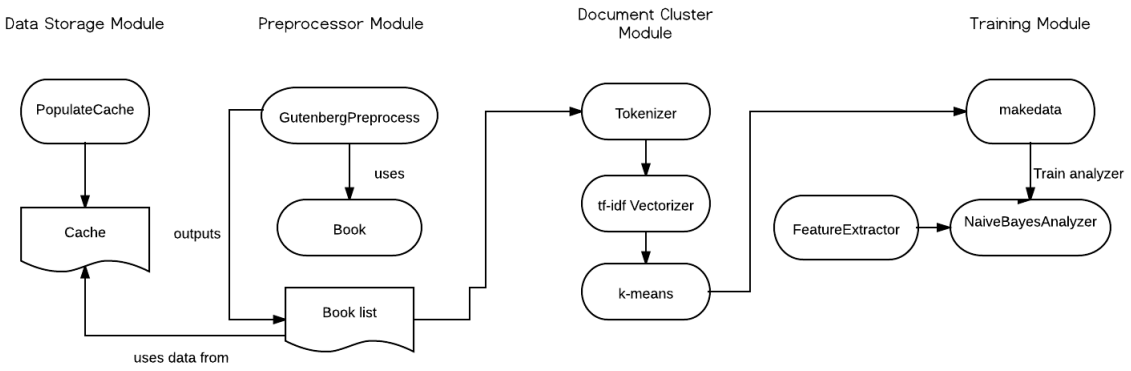

Figure 1: Data-flow diagram for sentiment classifier

- transforming the corpus into vector space using tf-idf
- calculating cosine distance between each document as a measure of similarity
- clustering the documents using the k-means algorithm

2 Architecture of the project

We now provide a high-level data-flow diagram of all modules and then provide sample codes on how we implemented them.

2.1 Data-flow diagram

Figure 1 depicts the DFD for our sentiment classifier. There are four modules. The data storage module is responsible for creating the cache on the local disk from the internet so that the Gutenberg APIs can access all data required from e-books. This is a one-off process and takes nearly 12 hours to complete. The preprocessor module uses this data to create instances of `Book`, and `pickle`s the list into local storage, denoted in the diagram by `Book list`. The document cluster uses this file to process the text into vectors and cluster them appropriately so that they can be used by the Naive Bayes classifier in the training module.

2.2 Data storage module

The following code populates the metadata cache.

Listing 1: Populate metadata cache

```
def populate_cache():
    """ Populates metadata cache to work with gutenberg api"""
    from gutenberg.acquire import get_metadata_cache
    cache = get_metadata_cache()
    cache.populate()
```

2.3 Preprocessor module

We use the following definition (in Listing 2) of `Book` for our purposes. We may extend this class to have more features later.

Listing 2: `Book` class

```
class Book:
    def __init__(self, book_id, title="", content=""):
        self._id = book_id
```

```
        self._title = title
        self._content = content
```

Then, to process each file, we need to download the text and metadata from cache and update our list. We define a function `init_books` for this purpose, as in Listing 3. We first create our list and then save it in binary format into a file so that the object can get stored as is.

The function takes two parameters, a list of authors whose books should be downloaded, and a file name where we save the data at the end.

Listing 3: Create the book list

```
def init_books(author_file, json_file):
    """initialize book list with texts and save it to disk"""
    with open(author_file) as f:
        authors = list(f)

    authors = [i.strip() for i in authors]

    books = []
    for author in authors:
        s = get_etexts('author', author)
        for i in s:
            try:
                title, etext = list(get_metadata('title', i))[0],
                    strip_headers(load_etext(i)).strip()
                b = Book(i, title, etext)
                books.append(b)
            except UnknownDownloadUriException:
                # this book does not have a load_etext corresponding to it.
                pass

    with open(json_file, 'wb') as f:
        pickle.dump(books, f)
```

2.4 Document cluster module

The rest of this document explains in detail the document cluster module. Section 3 describes the stemming and tokenization, section 4 describes how to perform tf-idf and document similarity calculations, and in section 5 we explain the k-means clustering as applied to this problem.

3 Stop words, stemming and tokenization

The first step in the clustering process is to tokenize the data. We use NLTK's Snowball Stemmer as this gives better results than Porter's stemmer. We create a list of stemmed words and tokenized words, to easily map one to the other, as the stemmed word is representative of all such words. Listing 4 shows a snippet of how we achieve this.

Listing 4: Tokenizing and stemming the books

```
def tokenize_data(book_text):
    # load nltk's English stopwords as variable called 'stopwords'
    stopwords = nltk.corpus.stopwords.words('english')

    from nltk.stem.snowball import SnowballStemmer
    stemmer = SnowballStemmer("english")
```

```
C:\Users\anind\Anaconda3\python.exe C:/Users/anind/OneDrive/Documents/GitHub/GutenbergSentiment/clustering.py
            words
a               A
christma  CHRISTMAS
carol       CAROL
in             IN
prose       PROSE
```

Figure 2: Output for Pandas dataframe

```
    vocab_stemmed = []
    vocab_tokenized = []
    for i in book_text:
        vocab_stemmed.extend(tokenize_and_stem(i, stem=True, stemmer=stemmer))
        vocab_tokenized.extend(tokenize_and_stem(i))

    return vocab_tokenized, vocab_stemmed

def tokenize_and_stem(text, stem=False, stemmer=None):
    tokens = [word for sent in nltk.sent_tokenize(text) for word in nltk.word_tokenize(sent)]
    filtered_tokens = []
    # filter out any tokens not containing letters (e.g., numeric tokens, raw punctuation)
    for token in tokens:
        if re.search('[a-zA-Z]', token):
            filtered_tokens.append(token)
    if stem:
        stems = [stemmer.stem(t) for t in filtered_tokens]
        return stems
    return filtered_tokens
```

As we see, the same function is used for both tokenizing and stemming based on the parameter `stem`. The result from `tokenize_data` can then be loaded into `pandas` dataframe to visualize it better, with the stemmed vocabulary as the index and the tokenized words as the column. The benefit of this is it provides an efficient way to look up a stem and return a full token. We created a pandas frame and printed the first five lines of the frame as follows:

Listing 5: Pandas dataframe

```
vocab_frame = pd.DataFrame({'words': vocab_tokenized}, index=vocab_stemmed)
print(vocab_frame.head())
```

The output was as expected, shown in Figure 2.

4 Tf-idf and document similarity

4.1 Definition

From Wikipedia, in information retrieval, tf–idf or TFIDF, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus.

4.2 tf-idf Vectorizer

We have used the tf-idf vectorizer to compute document similarity. The parameter to the function is a list of e-book contents. Some of the tweaked parameters include that we use unigrams, bigrams and trigrams; and that the minimum number of times a word should occur is in at least 20% of the document. The function is as below:

Listing 6: Tf-idf vectorizer

```
def tf_idf_calc(synopses):
    from sklearn.feature_extraction.text import TfidfVectorizer
    tfidf_vectorizer = TfidfVectorizer(max_df=0.8, max_features=200000,
                                       min_df=0.2, stop_words='english',
                                       use_idf=True, tokenizer=tokenize_and_stem, ngram_range=(1, 3))

    tfidf_matrix = tfidf_vectorizer.fit_transform(synopses) # fit the vectorizer to synopses

    terms = tfidf_vectorizer.get_feature_names()

    from sklearn.metrics.pairwise import cosine_similarity
    dist = 1 - cosine_similarity(tfidf_matrix)

    return tfidf_vectorizer, tfidf_matrix, dist, terms
```

5 K-means clustering

With the tf-idf matrix calculated, we can now run clustering algorithms on our documents. The k-means is an iterative algorithm. In this section, we explain k-means and how it was applied to our documents.

5.1 Algorithm

The goal of the algorithm is to **minimize distortion measure** by alternative optimization between $\{r_{nk}\}$ and $\{\mu_k\}$.

Algorithm 1 K Means

1: **Step 0** Initialize $\{\mu_k\}$ to some values.
2: **Step 1** Assume the current value of $\{\mu_k\}$ fixed, minimize J over $\{r_{nk}\}$, which leads to the following cluster assignment rule.
3:

$$r_{nk} = \begin{cases} 1 & \text{for the closest cluster k to n} \\ 0 & \text{otherwise} \end{cases}$$

4: Recompute the means and repeat until converged.

5.2 Implementation in our code

We found it took several runs for the algorithm to converge a global optimum as k-means is susceptible to reaching local optima. Below is the code that we implemented for k-means. Since this is a long process, we do it once and then save the data into a dump file for future usage.

Listing 7: K-means

```
def k_means(tfidf_matrix):
    from sklearn.cluster import KMeans
    num_clusters = 5
    km = KMeans(n_clusters=num_clusters)
    km.fit(tfidf_matrix)

    from sklearn.externals import joblib
    joblib.dump(km, 'doc_cluster.pkl')

    return km
```

```
0    159
1    119
2    116
4     72
3     20
Name: cluster, dtype: int64
```

Figure 3: Cluster output

5.3 Outputs

The k-means for our code divides the books into 5 clusters as mentioned in the code. The output we received was asymmetric, and is because the data downloaded from the Gutenberg has some non-English authors right now. We are trying to optimize the book-list and author-list to get more accurate results before we can send this result to the classifier. The output is shown in Figure 3.

6 Next Steps

6.1 Identifying top words

Identifying top words is a challenge because currently some small words, as well as words in different languages are coming up as most frequently used words. We are developing an algorithm that can give better results on the top words in each cluster, which can then be used to tag the e-books in that cluster.

6.2 Visualizing the result

We are also working on plots to help visualize the data, so that the clustering algorithms becomes more understandable to us.

6.3 Going back to Naive Bayes

Once the clusters are centered on their most important words as in 6.1, we can apply Naive Bayes to correctly classify them and use a test-set to check our results.