Attn: Dr. Sun Aixin

AI6122 Text Data Management and Processing

We hereby declare that the attached group assignment has been researched, undertaken, completed and submitted as a collective effort by the group members listed below. We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work. We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work.

Name	Signature / Date
Li Kaiyu	李凯宇 2022/10/20
Chen Lei	Chen Lei 2022/10/20
Chen Yueqi	Chen Yueqi 2022/10/20
Chang Lo-Wei	Chang Lo-Wei 2022.10.20
Li Jiayi	Li Jiayi 2022/10/20

Important note:

Name must **EXACTLY MATCH** the one printed on your Matriculation Card. Any mismatch leads to **THREE (3)** marks deduction.

AI6122: Text Data Management & Analysis Group Project Report

Li Kaiyu
Nanyang Technological University
Singapore
S220048@ntu.edu.sg

Chen Lei
Nanyang Technological University
Singapore
chen1552@e.ntu.edu.sg

Li Jiayi
Nanyang Technological University
Singapore
JLI105@e.ntu.edu.sg

Chen Yueqi
Nanyang Technological University
Singapore
ychen118@e.ntu.edu.sg

Chang Lo-Wei
Nanyang Technological University
Singapore
lchang009@e.ntu.edu.sg

ABSTRACT

In this report, we conduct a series of experiments and develop a series of software applications with the Amazon product dataset. Our goal is to study text management and processing related techniques, as well as leveraging these techniques to complete a set of NLP and IR tasks.

KEYWORDS

datasets, neural networks, gaze detection, text tagging, tokenization, stemming

THIRD-PARTY LIBRARIES

This is a declaration of all natural language processing and information retrieval third-party libraries used in this project. Corresponding to each section, the following libraries are used:

- Data Analysis: NLTK[3], Spacy[7], WordCloud[11]
- Search Engine: NLTK, Scipy[14]
- Review Summarizer: NLTK, Scipy, Scikit-Learn[13], PKE[4]
- Application: NLTK, Textblob[10], Surprise[9]

1 DATASET ANALYSIS

1.1 Dataset Overview

Our datasets are derived from the Amazon Review Data[6], which consists of 142.8 million product reviews and metadata from Amazon, from May 1996 - July 2014. We choose two datasets, **Digital Music** and **Kindle Store**, among all 24 datasets provided with 5-core filtering(each user or item has at least 5 reviews). Then, we randomly sample 200 products from both datasets and leave the datasets with all reviews of these 400 products. Any subsequent analysis and development is based on these two sampled datasets, with 200 products in each.

1.2 Writing Style

First, we look into the writing style of the two chosen datasets. Several reviews are drawn from each dataset and compared with a paragraph drawn from a news article published by Strait Times[5]. Figure 1 and figure 2 show some segments from the reviews and the news article respectively. Each blue underline in these figures indicates an occurrence of incorrect grammar. Each red underline in these figures indicates an occurrence of a wrongly spelled word. Apparently, compared with the news article, Amazon product reviews are likely to have more grammar or spelling errors, since

> This album showcased the versatility of the late great BIG. It truly makes you wonder what the hip hop world would be like had we still been able to line up at stores for his next release. He set the tone for all albums with this one. Every artist to this day is still trying to duplicate this winning formula. BIG had asong on this album for each and every individual in the world. By far the best Hip Hop album I have ever expierienced.
>
> If your looking for a good quick fix.. A not so bad mini story with a touch of erotica that can make your eyebrows raise a little and go oh.. Yup this is the book for you!Luke is a wealthy man.. Indulging on his fantasy for one time a year at a place in New Orleans that's a hotel which transforms into something else completely.. He has been going to this place for years with his eyes set on one woman.. And he can't have her.. Seraphina is a whipper snapper.. Takes no prisoners and lives and works by the rules.. Luke wants her.. She wants him but doesn't "cross that line" until one night.. Which turns into two and then they are forced to make a decision... I liked it.. Great book! I'm already onto the next one..
>
> As brilliant and warm as the Alexander family themselves. I loved it! Can't wait to read the continuing stories of this family. Thanks for the fun read Minx, and well done. Very well done indeed!!!

Figure 1: Reviews Sampled from Datasets

> Although clinical studies for the latest bivalent Cominarty vaccine are still underway, HSA said a previous study - for the bivalent Pfizer-BioNTech vaccine that protects against the original strain as well as the BA.1/2 variants - showed promising results.
>
> In that study, the bivalent vaccine was shown to elicit a stronger immune response against the Omicron BA.1 variant, while still being effective against the original Covid-19 strain.
>
> HSA said it used this previous study as the primary basis for granting approval, as Omicron sub-variants are closely related to one another.
>
> Under the Pandemic Special Access Route (PSAR), HSA is allowed to start evaluating new vaccines, medicines and medical devices from the early stages of clinical studies, instead of waiting for full data sets to be submitted.

Figure 2: Segment from News Article by Strait Times

they are more informal, written by ordinary people instead of professional new writers. There are also some special characters in the original views - for example, the '&#34' symbol appears twice in the second review in figure 1. Another interesting finding is that, the review text in our datasets tend to use more expressive symbols

and shorter sentences - for instance, the symbol '..' and '!!!' appears several times in the chosen reviews.

1.3 POS Tagging

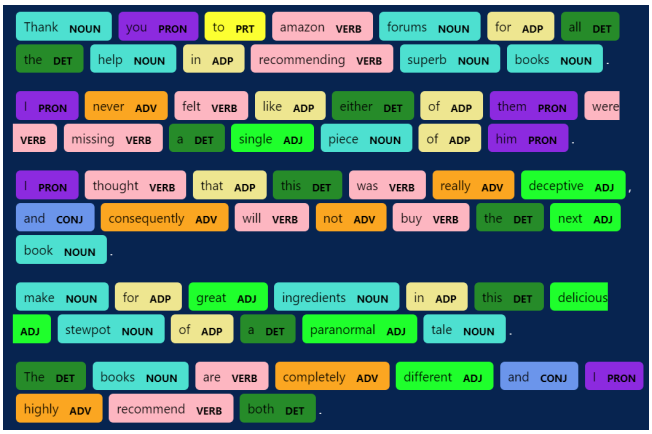

Figure 3: POS Tagging Examples

In this section, we evaluate the POS tagging results on the datasets. We randomly select 5 sentences from the datasets, apply POS tagging on them, and finally visualize the tagging result in figure 3. The overall POS tagging accuracy is good, but the tagging fails in some circumstances. The result of the 1st sentence illustrates that unseen words like 'amazon' may confuse the POS tagger, where the tagger wrongly tags 'amazon' as a verb, but tags 'Thank' as a noun. The 4th sentence demonstrates that incorrect grammar may also confuse the POS tagger, where the subject of sentence is missing, and the verb 'make' is wrongly tagged as a noun.

1.4 Sentence Segmentation

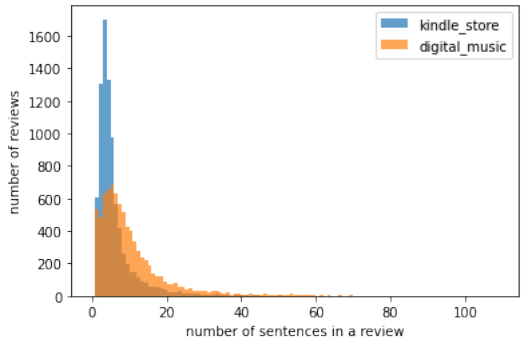

Figure 4: Sentence Segmentation Histogram

Figure 4 and figure 5 exhibits the distribution of sentence quantity of each single review. The bin width of the histogram is 1. From both figures, the reviews of the Digital Music dataset have a broad distribution and have more sentences in average. In comparison, reviews of Kindle Store have a sharp distribution with a smaller mean value. Every review has 1 sentence at least, so vacant reviews

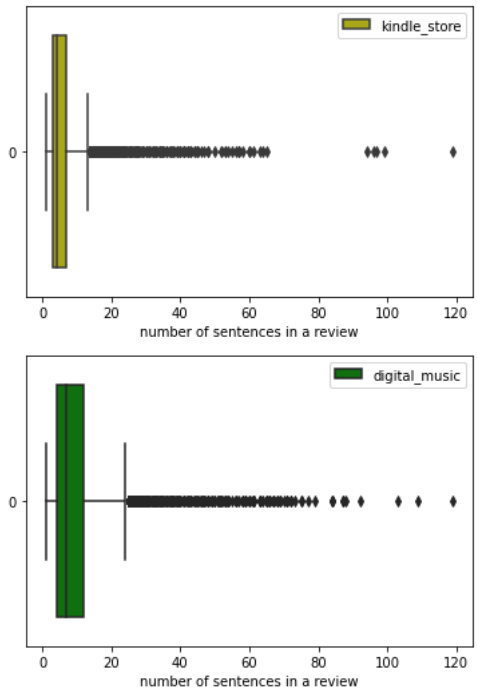

Figure 5: Sentence Segmentation Boxplot

are prohibited. Note that both datasets have some outliers which have significantly more sentences than the average, resulting in wide ranges of review length in terms of number of sentences. We further examined a few reviews from those outliers, and it turns out that long reviews do not necessarily carry more information - many of them have a large segment directly quoting the book or the song.

1.5 Tokenization and Stemming

Figure 6 illustrates the distribution of number of tokens in each review in a histogram with bin width equal to 40. Compare figure 6 with figure 4, we can observe that the distribution of review length in tokens of both datasets follows the same pattern as their distribution of review length in sentences. That is, the Digital Music dataset has a broader distribution with higher mean value, and the Kindle Store dataset is on the contrary. This result implies that the sentences in the two datasets have similar length in terms of number of tokens.

We now investigate the relative frequency of tokens. We compute the collection of unique tokens in each dataset, and sort them by the times that they appear in the dataset, in descending order. To study the effect of stemming, we then perform stemming on both datasets and compute the distributions again. Note that the **Snowball Stemmer** provided by NLTK is applied. The two sets of distributions are shown in figure 7, in which the result without stemming is presented in solid lines, the other in dashed lines. We also applied log 10 function to both x-axis and y-axis. Every result curve roughly follows a straight line, which confirms the Zipf's law - The i-th most frequent token has frequency proportional to

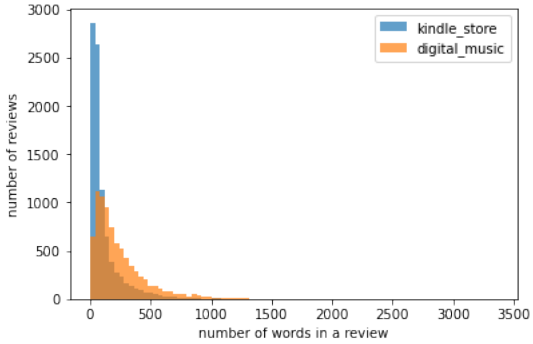

Figure 6: Word Tokenization Histogram

$\frac{1}{i}$. The result with stemming has a similar distribution in terms of high-ranked tokens, but its middle-ranked tokens have higher frequency, and it has less amount of low-ranked tokens. This is because the stemming process merges inflected tokens to a single token, known as their word stem. As a result, stemmed datasets have less number of unique tokens, and their tokens are merged with other tokens or merged to generate a new token(e.g. story, stories -> stori). The high-ranked tokens with high frequencies are not influenced much, this may be due to the fact that a number of them are stopwords.

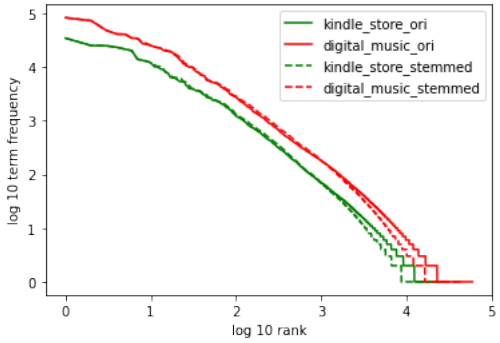

Figure 7: Token Frequency Plot (With Stemming and Without Stemming)

1.6 Indicativate Words

In this section, we want to find out the indicative words of each dataset. First, we should clarify what is an indicative word. On the one hand, an indicative word should be representative of the whole dataset, instead of just covering a subset of the dataset. For example, for the Kindle Store dataset, the word 'book' should be better than any book title. On the other hand, an indicative word should distinguish its corresponding dataset from other datasets. We measure the indicativeness using pointwise relative entropy. That is, if $P(w|D_1), P(w|D_2)$ are the probabilities of observing word w in dataset D_1 and D_2, then the relative entropy of the word is given by:

$$D_{KL} = P(w|D_1)log(\frac{P(w|D_1)}{P(w|D_2)}) \quad (1)$$

Where a higher result value in equation (1) means the word w is more indicative for dataset D_1. We can also compute the indicativeness of word w for D_2 vice versa. However, in practice, the denominator in equation (1) is possible to be zero. One solution is to implement a laplace(add k) smoothing to solve the issue. However, we apply a different approach here - we only consider the words that exist in both datasets. Because we are looking for more general or abstract words that are representative for the whole dataset(e.g. 'book' vs 'Harry Potter', 'Music' vs 'Despacito'), and those words are likely to exist in every dataset. We also remove all stopwords included in the English stopwords set provided by NLTK. The following word clouds show the top-30 most indicative words of both datasets.

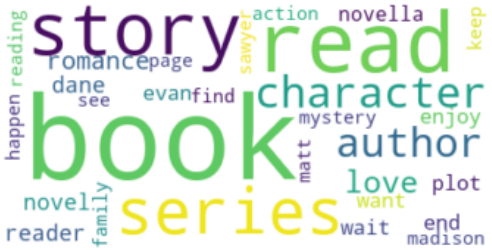

Figure 8: Top 30 Most Indicative Words of Kindle Store

Figure 9: Top 30 Most Indicative Words of Digital Music

We select the top-10 most indicative words from the word clouds and list them below:

- **Kindle Store**: book, read, story, series, character, author, love, romance, reader, novel.
- **Digital Music**: album, song, track, music, lyric, sound, listen, band, rap, record.

2 DEVELOPMENT OF A SIMPLE SEARCH ENGINE

2.1 Main idea of the simple search engine

This simple search engine supports searching in the following format but the term order is interchangeable, where * represents 0 or more occurrences of the preceding terms:

reviewerID*	**asin***	plain-text*

The output would be the **Top N** (N is configurable) ranked relevant reviews with its scores, docID, and snippets whenever possible.

Below shows the basic idea of our search engine:

(1) Consider each review as a document. Generate two sub-documents, *IDdoc* and *textdoc*, for four searchable fields we choose. *IDdoc* contains fields **reviewerID** and **asin**; *textdoc* contains fields **reviewText** and **summary**.
(2) Apply necessary preprocessing steps to *IDdoc* and *textdoc* depending on their data structures.
(3) Indexing *IDdoc* and *textdoc* differently as needed.
(For the following steps (4) (6), check if there are already $\geq N$ documents at each step. If so, rank these documents and return the top N results.)
(4) Check through *IDdoc* to get m_1 ($0 \leq m_1 \leq N$) documents containing both **reviewerID** and **asin** appeared in preprocessed query (if any). Rank them with respect to plain text in query (if any) and store the ranked results, then go to step (5).
(5) Check through *IDdoc* to get m_2 ($0 \leq m_2 \leq N - m_1$) documents containing either **reviewerID** or **asin** appeared in preprocessed user query (if any) and store the ranked results, then go to step (6).
(6) Check through *textdoc* to get documents containing plain text (single keyword/phrase query) appeared in preprocessed user query (if any).
  6.1 if none, return the m_1 results or (m_1+m_2) results obtained in step (4) or step (5).
  6.2 Otherwise, if the plain text has k tokens, use a query parser to get phrases with k, $k-1$, $k-2$, ..., 1 tokens. Collect documents with respect to these phrases in decreasing order of the number of tokens until N or $(N - m_1)$ or $(N - m_1 - m_2)$ documents are collected. Rank documents containing same number of tokens only and append the ranked documents containing fewer number of tokens after those containing more.

Note: ranking is based on tf-idf weighting and cosine similarity

2.2 Document Collection

This simple search engine supports searching in fields: **reviewerID**, **asin**, **reviewText**, and **summary** for different searching purposes. Searching for a specific **reviewerID** or a **asin** will result in all the reviews written by this reviewer or all the reviews under the corresponding product respectively. Our search engine also supports single or phrase queries in plain text fields, including **reviewText** and **summary**. In addition, it is also possible to simultaneously search among all these four fields. For example, the result of searching "A3EBHHCZO-6V2A4 5555991584 memory of trees" is all the reviews containing keyword "memory of trees" under product <5555991584> written by reviewer <A3EBHHCZO6V2A4>.

Considering the different formats and meanings of these four fields, we store them into two lists. Fields **reviewerID** and **asin** in list *IDdoc*, while **reviewText** and **summary** in list *textdoc*. Indexes in two lists are corresponding to the document indexes. Here, each review is a document, i.e.

- *IDdoc[i]* = [reviewerID, asin] represents the **reviewerID** and **asin** of document i.
- *textdoc[i]* = reviewText + " " + summary represents the **reviewText** and **summary** of document i.

The benefits of setting two lists *IDdoc* and *textdoc* will be shown later in section2.3 (data processing), section 2.4 (indexing), and section 2.6 (ranking and score).

2.3 Data Processing

Depending on the data structure of two lists we collect, different data processing methods should be applied. Since **asin** and **reviewerID** is encoded with capital letters and digits (e.g. A3EBHHCZO6V2A4), only case-folding is required for *IDdoc*. However, for plain text **reviewText** and **summary**, *textdoc* needs case-folding, tokenization, NLTK stopwords removal, punctuation removal, and Porter stemmer and the order matters. Removing stopwords after tokenization and case-folding is necessary and stemming should be applied at last. As shown in Table 1, *IDdoc* needs much fewer processing methods than *textdoc*, setting as two lists would be helpful for efficiency.

Table 1: Choice of linguistic processing

	IDdoc	textdoc
case-folding	✓	✓
tokenization		✓
stopwords removal		✓
punctuation removal		✓
Porter stemmer		✓

2.4 Indexing

Depending on the data structure of *IDdoc* and *textdoc*, we index them differently. Same as **reviwerID**, every **asin** in *IDdoc* is only one token. Thus, there is no need to store their positional indexes and inverted index is enough for *IDdoc*. However, in order to deal with phrase query, it is necessary to use positional indexes for plain text in *textdoc*. Scanning through all documents once is enough to index both *IDdoc* and *textdoc*. Figure 10 shows the implementation of indexing *IDdoc* and *textdoc* differently at the same time.

Figure 11 shows a plot of the time needed to index every 10% of documents. As shown in the plot, the index time ranges from 0.05s to 0.25s per 10% of documents and it is acceptable. It is noticeable that the 8_{th} (80%) and 10_{th} 10% (100%) documents required index time more than 0.2 seconds. The reason here is that longer documents fall in these two groups and they need more time to finish positional index.

2.5 Query Parser

Query parser is only used for plain text in *textdoc* in this search engine. Basically, the main idea of this query parser is to allow a phrase query to spawn one or more queries to the indexes and single keywords can be considered as a special case of phrase query with only one token. Firstly, run the original query with k tokens as phrase query and obtain documents containing this query. Then we run the two sub-phrase queries with $(k-1)$ tokens. Iterates the number of tokens of phrase queries until enough documents are

```python
# Index
def index(doc, IDdoc):
    ind = {}
    id_collection = {}
    for i in range(len(doc)):
        if i%(int(len(doc)/10)) == 0:
            start = time.time()
        for j in range(len(doc[i])):
            if doc[i][j] not in ind.keys():
                ind[doc[i][j]] = [1]
                ind[doc[i][j]].append({})
                ind[doc[i][j]][1][i] = [j]
            elif i not in ind[doc[i][j]][1].keys():
                ind[doc[i][j]][0] += 1
                ind[doc[i][j]][1][i] = [j]
            else:
                ind[doc[i][j]][1][i].append(j)
        if IDdoc[i][0] not in id_collection.keys():
            id_collection[IDdoc[i][0]] = [i]
        else:
            id_collection[IDdoc[i][0]].append(i)
        if IDdoc[i][1] not in id_collection.keys():
            id_collection[IDdoc[i][1]] = [i]
        else:
            id_collection[IDdoc[i][1]].append(i)
        if (i+1)%(int(len(doc)/10)) == 0:
            print("%s%% takes %.3f seconds" % \
                  (int((i+1)*10/(int(len(doc)/10))), \
                   time.time() - start))
    return ind, id_collection
```

Figure 10: Code of indexing

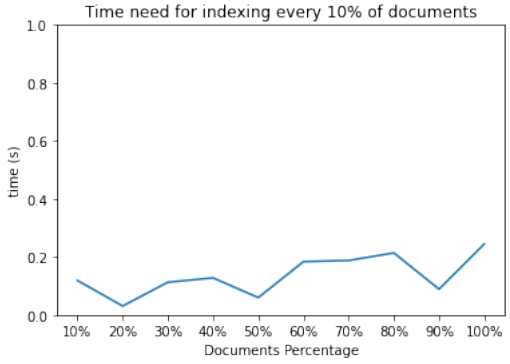

Figure 11: Plot of indexing time

found or until the number of tokens of phrase queries is reduced to 0. In our algorithm, documents containing longer phrase query will have a higher priority than the shorter ones. Figure 12 shows the implementation of calculating the length of the longest phrase that each document contains.

2.6 Ranking and Score

Each document and query will be represented by a real-valued vector of tf-idf weights and cosine similarity will be used to measure the similarity between query vector and document vectors. Based on this high level idea, we apply some variants. We use a variant of tf-idf weighting scheme to improve efficiency. According to the table shown in Figure 13, we use *lnc.ltc* for *document.query*. This scheme allows our algorithm to leave off the idf weighting on

```python
def check(lis, query):
    k = 0
    while len(lis) > 0:
        newlis = {}
        for i in range(len(query)-1):
            if query[i] not in lis.keys():
                continue
            if query[i+1] not in lis.keys():
                continue
            for pos in lis[query[i]]:
                if pos+1 in lis[query[i+1]]:
                    if query[i+1] not in newlis.keys():
                        newlis[query[i+1]] = [pos]
                    else:
                        newlis[query[i+1]].append(pos)
        lis = newlis
        k += 1
    return k
```

Figure 12: Code of query parser

documents resulting in the efficiency improvement. The reason here is that knowing how rare every term is in every document is unnecessary and only the rarity of each term in query matters.

Term frequency		Document frequency		Normalization	
n (natural)	$tf_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(tf_{t,d})$	t (idf)	$\log \frac{N}{df_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2+w_2^2+...+w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times tf_{t,d}}{\max_t(tf_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N-df_t}{df_t}\}$	u (pivoted unique)	$1/u$
b (boolean)	$\begin{cases} 1 & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/CharLength^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1+\log(tf_{t,d})}{1+\log(ave_{t \in d}(tf_{t,d}))}$				

Figure 13: table of tf-idf weighting variants

2.6.1 ranking scheme. Based on the query, we can filter and classify documents into at most three priority classes: (1) 1^{st} priority: documents contain both **reviewerID** and **asin** in the query (if any); (2) 2^{nd} priority: documents contain either one (if any); (3) 3^{rd} priority: documents only contain plain text. To rank documents within these priority classes, we will check the plain text. Documents with longer phrases will have a higher priority than shorter ones and documents containing more terms will have higher priority than those containing less ones. For documents in the same category with the same length of phrase, we will compare their cosine similarity according to the tf-idf invariant we mentioned above.

2.6.2 score scheme. In our algorithm, the score for ranking documents with respect to the query is not only the cosine similarity between the query vector and document vectors. Based on *IDdoc* and query parser, the documents will be added different scores.Generally, more scores will be added to documents containing **reviewerID**, **asin**, and phrase query with more tokens.

Suppose a user query consists of any of reviewerID, asin, and plain-text. $\vec{q}$ is a phrase with k tokens obtained from plain-text by query parser, $\vec{d}$ is the document vector based on *textdoc*, and i is the number of ID's (reviewerID, asin) in document matched with ID's in user query. Then the score function would be:

$$(k+1) \times i + k + cos(\vec{q} \cdot \vec{d})$$

2.7 Results Analysis

Figure 14 shows the results of searching **reviewerID**, **asin**, and plain text with the content of "AKAOYM48DSUPL B00003002C jazz pop". Since documents containing all elements will have the highest priority, the first result clearly proved this priority rule. Results 2 10 proves the second highest priority of documents containing either **reviewerID** or **asin** since all **asin** = "B00003002C" here.

```
Search for: (type "q" to quit) AKAOYM48DSUPL  B00003002C jazz pop
   Rank  DocID      ReviewerID        asin                                                  Snippets     Score
0     1  12633  AKAOYM48DSUPL  B00003002C  ... last two albums. Jazz, Pop, and Disco dist...  5.219651
1     2  12650  A1GN8UJIZLCA59  B00003002C  ... synthesis of their jazz, pop and rock mixt...  2.185102
2     3  12631  A2PH5Z4M4UZWOL  B00003002C  ... timeless melodies, sophisticated jazz and ...  2.157181
3     4  12618  A3MQVJ1JNGHDSP  B00003002C  ... diverse styles from jazz, to pop, rock, R&...  2.139790
4     5  12655  A3MR1REKDWAQSV  B00003002C  ... time album. For jazz purists, Steely Dan m...  1.258453
5     6  12673  A339IE87G18AJ1  B00003002C  ... combine elements of jazz, rock, and pop bu...  1.147083
6     7  12616  A2UYAFQ40U2PHS  B00003002C  ... Steely Dan. There's jazz, rock, blues, pro...  1.146468
7     8  12672  A23IOP10D1OG7V  B00003002C  ... need more elegant pop music in this vein. ...  1.142173
8     9  12647  A38CJW1RNUHMN6  B00003002C  ... I remember each pop,rock,jazz, fusion fill...  1.132597
9    10  12636  A3LZGLA88K0LA0  B00003002C  ... Steely Dan as pop's essential jazz/rock in...  1.127066
Search takes 0.448 seconds
```

Figure 14: reviwerID + asin + plain text

Figure 15 shows the result of query with multiple ID terms and the result is also satisfactory here. First two results contain each pair of ID in query. In addition, the result also proves that the order of terms in query will not influence the searching as the query follows a format of [reviwerID asin reviwerID asin plain-text] instead of [reviwerID* asin* plain-text].

```
Search for: (type "q" to quit) A328AACLGP6ZI3  B000000OQS A1D2C0WDCSHUWZ  B000NPE7YC jazz pop
   Rank  DocID      ReviewerID        asin                                                  Snippets     Score
0     1  16641  A1D2C0WDCSHUWZ  B000NPE7YC   ... that draw from jazz, pop, folk, a bit of ...  5.155606
1     2   9013  A328AACLGP6ZI3  B000000OQS  ... You gotta love this one! Joni's Jazz / Pop...  2.168365
2     3  16658  A2KEKPMI3WQMPA  B000NPE7YC  ... she's got her jazz cred half originals hal...  1.126718
3     4  16632   ANCOMAI0I7LVG  B000NPE7YC  ... her in his Jazz magazine. `Reminder' was t...  1.063342
4     5  16643   A5SC9O8TZJ6DK  B000NPE7YC  ... nova, folk and jazz elements heard promine...  1.058401
5     6  16631  A2PV6GK1HV54Y9  B000NPE7YC  ... I,2,3,4 is just pop dribble. People may ad...  1.011709
6     7  16666  A1W3MBU3V85BG9  B000NPE7YC  ... the kind of pop that (regardless of the iP...  1.011531
7     8  16656  A24N1BAS3CU27H  B000NPE7YC     ... is a snappy pop number. My CD comes ...  1.008474
8     9  16949  A1D2C0WDCSHUWZ  B002TZVH2Y  ... languid, vaguely dreamlike pop, full of su...  1.007792
9    10  16613  A1D2C0WDCSHUWZ  B000NA27TE  ... -- stately tambourine pop, an acoustic ind...  1.007465
```

Figure 15: reviwerID + plain text

3 DEVELOPMENT OF A REVIEW SUMMARIZER

This section will describe the design and implementation of a review summarizer developed by our team. First we will talk about the ideal summarizer in our opinion and the technical challenges we've met so far. Then we will describe the way we process the data for developing the summarizer.After that we will mainly explain the RAKE model and some other ones used to implement the summarizer. Last the result of each summarizer will be given with comparison.

3.1 The Ideal Summarizer and Challenges

The reviews by users give the users' feelings after using the particular products, the overall evaluations and sometimes the reviews will recommend or not recommend for other users. However, some products may have a huge amount of reviews.It will take users a long time to read them and can hard understand the features of the product. Besides, different users may have various evaluations for the same product which may make users confused about the products. So we think a good summarizer need to contain the following features:

- Get the main point of the product (Topic phrases).
- The summary should be concise phrases as far as possible, making it easy to understand (Detail phrases).
- Reflect the most common users' evaluation for the product(Sentiment phrases).

Based on the features above, we think the combination of some phrases can be a good summarizer. It should contain phrases providing the features of the product, users' sentiments, recommendation,feeling about the products or just some description on the details of products. And those phrases should be as common as possible.

During the work we've met many challenges. One of them is how to let the computer know the language because the machine can't understand the natural language, especially the relation between each word, so we try to directly extract the origin phrases from the reviews. On the other hand, the way to evaluate the models' performance is hard to decide since it can't be judged by some scores. We need to find more objective methods to evaluate the performance and the questionnaire may be a good choice.

3.2 Data Processing

To prepare the dataset, we merge all the review text of the same product. We iterate all the sample review data and merge the ones with the same *asin*, which is the product ID, and finally create a dictionary where the key is the *asin* and value is the review. After merging the reviews, every review will firstly be segmented into single sentences.

For every single sentence from reviews, normalizing word form is necessary. We remove all the punctuation and make every letter lowercase since the punctuation and whether the letter is uppercase or lowercase don't affect tokenization and normalization a lot. Then we will remove all the stop words, which refer to the insignificant words in NLP. In this section we leverage the stopword package from both NLTK and Spacy, and merge them together, resulting in 383 stop words in total. The count of the stop words is shown in the table2.

Table 2: The count of stop words

package	count
nltk	180
spack	326
merge	383

After that we look for the POS tag of words, thus we can do the lemmatization to transfer every word into their lemma with their tags.

At last we tokenize the processed sentences, split them into single words and count the frequency of each word in descending order. The result will be stored in a dictionary as the following form. Figure16 shows the data pre-processing before we realize a summarizer.

$$\{asin : \{word : frequency\}\} \tag{2}$$

```python
def data_preprocess(original_review, stop_words):
    # to lowercase
    review = original_review.lower()
    # remove punctuation
    review = re.sub(r'[^\w\s]', ' ', review).strip()

    lemmatizer = WordNetLemmatizer()
    review_words = review.split(' ')
    review_words_tag = nltk.pos_tag(review_words)
    processed_review = ""
    # remove stop words&lemma
    for word, tag in review_words_tag:
        wntag = tag[0].lower()
        wntag = wntag if wntag in ['a', 'r', 'n', 'v'] else None
        if not wntag:
            lemma_word = word
        else:
            lemma_word = lemmatizer.lemmatize(word, wntag)
        if lemma_word not in stop_words:
            processed_review = processed_review + lemma_word + " "
    processed_review = " ".join(processed_review.split())
    processed_review = processed_review.strip()
    return processed_review
```

Figure 16: Code of data pre-processing

3.3 RAKE: Rapid Automatic Keyword Extraction Algorithm

There are two types of summarization methods: Extractive summarization and abstractive summarization. The former is to concatenate important sentences or paragraphs without understanding the meaning of those sentences, and the latter is to generate a meaningful summary based on learning. The system is of implementation of text document statistical and linguistic analysis. The normalization concept ensures that sentences are given weights only based on the substance of their words, rather than their word count. Due to the importance of the words it contains, even a brief but significant phrase is given due consideration. The algorithm may be further tuned by using language characteristics to get a higher level summary.[1]

After comparing the most popular and useful methods and techniques, we finally choose **the Rapid Automatic Keyword Extraction Algorithm (RAKE)** as the optimal efficient algorithm to develop the summarizer for review texts in our dataset.

3.3.1 The RAKE Algorithm. RAKE, as known as Rapid Automatic Keyword Extraction, is a very effective technique for extracting keywords and key phrases from individual documents that enables use in the dynamic collection. It is also quite successful at managing a wide variety of documents, especially the kind of writing that adheres to certain grammatical norms, and it can be adapted to new domains very quickly. RAKE is founded on the fact that keywords usually include numerous words with stop words or normal punctuation, or we can use functional terms like "and," "of," "the," etc. with little to no lexical value.

A list of stop words, a set of phrase delimiters, and a set of word delimiters are included in the input parameters for the RAKE Algorithm. It divides the document into candidate keywords using stop words and phrase delimiters; these candidate keywords are mostly the terms that aid a developer in extracting the precise keyword required to obtain information from the page.[12]

As previously, it is known that RAKE parses the document by stop words and phrase delimiters to find the main content containing word as a candidate keyword. According to the data processing before, we have generated a merged set of stopwords and have segmented the review texts into sentences by their own product ID. Therefore, we can directly apply the obtained stopwords set and precessed texts into the RAKE algorithm, and get the candidate keywords and key phrases with scores.

In the RAKE algorithm, after obtaining all possible keywords, we can get a graph of word co-occurrence, through which the score for each keyword can be calculated by $Word\_Frequency$, $Word\_Degree$, and $\frac{Word\_Degree}{Word\_Frequency}$ as follows:

```python
def word_scores(phraseList):
    frequency = {}
    degree = {}
    for phrase in phraseList:
        wordList = split_words(phrase)
        wordListLen = len(wordList)
        wordListDegree = wordListLen - 1
        for word in wordList:
            frequency.setdefault(word, 0)
            frequency[word] += 1
            degree.setdefault(word, 0)
            degree[word] += wordListDegree
    for i in frequency:
        degree[i] = frequency[i] + degree[i]
    score = {}
    for i in frequency:
        score.setdefault(i, 0)
        score[i] = degree[i] / (frequency[i] * 1.0)
    return score
```

Figure 17: RAKE: Score of words

Gaining the scores for keywords and key phrases, the highest H candidate key words and phrases are chosen as the summarization. The H value is determined by the square root of the total number of candidate key words and phrases.

3.3.2 Baseline Methods. In this project, we select four popular and useful techniques in extracting keywords and key phrases as the baseline methods to evaluate our summarizer.

- TF-IDF: TF-IDF, short for term frequency - inverse document frequency, is using appearance count of a word in the document and the count of documents containing the word to calculate the TF-IDF score of the word.
- TextRank: Based on Google PageRank, TextRank is to generate sentence vector similarity graphs for calculating scores and ranks.
- YAKE!: A feature-based system for keyword and key phrase extraction.
- TopicRank: Improved TextRank method.

3.3.3 Results and Evaluation. After applying the RAKE algorithm and baseline models, we have obtained sets of key phrases generated by different algorithms. Take a product in Kindle Store with 1771 reviews as an example. Results, concrete analysis and verification of considering RAKE as the ideal summarizer are shown in Table 2 and Table 3.

Type of Phrases	Output	Comments
Topic	"time travel romances" "pre-war america"	This two phrases can give the buyer a rough idea about this book - a romantic time travel story happening in pre-war america.
Sentiment	"enjoy time travel books" "like time travel books"	The "enjoy" and "like" show that this book is readable and enjoyable.
Detail	"lovely young lady""rare planetary alignment"	These two phrases can let buyer learn about some details like the personal character and unique parts of this product.
Others	"21-year-old trailblazing grandmother" "northwest passage series""time travel element" "better time travel books"	Repeated Summaries

Table 3: Result Analysis of RAKE for a Kindle Store Example

Type of Phrases	Output	Comments
Topic	"hip-hop collection"	This two phrases can give the buyer a rough idea about this song or album - style of hip-hop by Jay-Z.
Sentiment	"nice dre-type beat" "favorite jay-z song" "hottest jay-z song" "best hip-hop album"	The "nice", "favorite", "hottest" and "best" show that this song is super good.
Detail	"roc-a-fella" "jay-z classic song"	This gives the record company, the singer, and it is a classic one.
Others	"hip-hop history" "hip-hop heads" "roc la familia"	Repeated Summaries

Table 4: Result Analysis of RAKE for a Digital Music Example

It can be seen that for the output of RAKE, it contains phrases about topic, which makes the buyer directly know if this product is he looks for, it contains phrases about sentiment, which makes the buyer learn about the overall evaluation about this product, and it contains phrases about details which help the buyer gain a more particular knowledge to understand whether the product can meet the buyer's requirement. Therefore, we initially consider RAKE to be the ideal summarizer.

To evaluate our RAKE algorithm by comparing with mentioned baseline models, each of teammates invited about 20 volunteers to finish a questionnaire for the results of four baseline models and the RAKE algorithm. Product 1 (2625 reviews) and Product 2 (1719 reviews) are in the Digital Music set and Product 3 (1771 reviews) and Product 4 (775 reviews) are in the Kindle Store set. We set 1 to 5 points for each summarization method to summarize the review texts of each product to let volunteers evaluate their overall degree of satisfaction with the intelligibility, practicability, and acceptability under comparison.

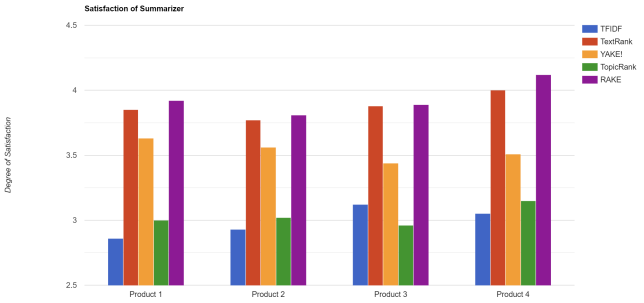

Figure 18: Statistical Results of Baseline Models and RAKE

As shown in Figure 1, we find that RAKE generally performs better than other alternative baseline models for all products. Although models like TextRank can also generate more or less appropriate summarized phrases, there exist small problems for them such as too many repetitive phrases or synonyms which make their phrase summaries not comprehensive. While for RAKE, for test products, it can stably cover relatively all types of phrases as Table 2. Besides, RAKE is one of the most efficient summarization methods. Test with a product with 2774 reviews, TextRank spends 3.44 seconds, YAKE spends 5.19 seconds, TF-IDF spends 4.80 seconds, TopicRank spends 53.78 second, but RAKE only spends 0.48 seconds, which is tens of times faster than other methods.

In conclusion, one of the reasons that RAKE can perform so well and stable is that it is domain-independent, it is fast and flexible for each single document without pre-training and easy to implement. However, the cons are also from RAKE's pros. It can not work on multi-documents at a time and it depends much on the quality of input of stopwords.

In the future, to keep the advantages as possible and reduce weaknesses, we will mainly work on combining RAKE with another supervised or reinforcement learning model to have a more wide-ranging algorithm model. Meanwhile, we may build an algorithm to auto classify outputs into types.

4 APPLICATION

In this section, we will introduce a recommender system based on sentiment analysis. It can recommend the products to the users who might be interested in some products, based on the review

texts and score the user has published and other users' with similar information. We use sentiment analysis to decrease the bias and noise overall and use a collaborative filtering system to realize the recommendation.

4.1 Sentiment Analysis

Sentiment analysis is a method to help discover the mood and emotions in the text. In this situation, sentiment analysis helps us understand how the user feels about the product. We can't just assume a user will recommend a product because sometimes the user may not be satisfied with it while giving a high overall. Here we use the *TextBlob* package to do the sentiment analysis. It is a python library for NLP, actively use NLTK to complete the task. From a whole review, we can get the following two parameters:

- **Polarity**: polarity defines the sentiment of the review, lying between -1 and 1 where -1 means negative,1 means positive and 0 means neutral.
- **Subjectivity**: subjectivity defines the amount of personal opinion and feelings, lying between 0 and 1. The more amount of personal opinion contained in the review, the higher subjectivity score is.

From the original review dataset, the polarity and subjectivity vs number of reviews are shown as the Figure19

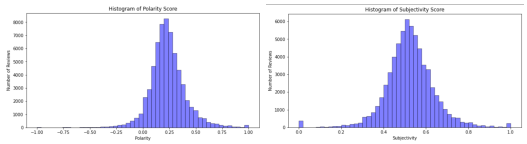

Figure 19: Histogram of polarity and subjectivity score

We can be informed from the above two figures that most reviews are positive and most reviews contain a certain amount of personal opinions, with the subjectivity between 0.4 and 0.6, and coincidentally, the histogram of subjectivity score nearly fit the gaussian distribution. So we decide to create a new score called *unbiased_overall*, which means the possibility the user may recommend this product, considering the user's review and personal subjectivity. It can be calculated by the following formula:

$$score_{unbiased\_overall} = overall + score_{polarity} * 5 * score_{subjectivity} \tag{3}$$

where *overall* means the original score the user gives to the product, $score_{polarity}$ means the polarity score, which was multiplied by 5 to keep the same weight as overall, and $score_{subjectivity}$ is the subjectivity score by the following rule as the subjectivity nearly fit the gaussian distribution:

$$score_{subjectivity} = \begin{cases} subjectivity * 2, if subjectivity <= 0.5 \\ 2 - subjectivity * 2, if subjectivity > 0.5 \end{cases} \tag{4}$$

where $subjectivity$ means the origin subjectivity score given by sentiment analysis. Figure20 shows how we do the sentiment analysis.

```python
from collections import defaultdict
from textblob import TextBlob
def get_sentiment(table):
    polarity = []
    subjectivity = []
    for idx, row in table.iterrows():
        review = TextBlob(row['reviewText'])
        sentiment = review.sentiment
        polarity.append(sentiment.polarity)
        subjectivity.append(sentiment.subjectivity)
    new_table = table
    new_table.insert(len(table.columns), column='polarity', value=polarity)
    new_table.insert(len(table.columns), column='subjectivity', value=subjectivity)
    return new_table
```

Figure 20: Code of sentiment analysis

4.2 Collaborative Filtering System with Sentiment Data Assisted[2]

A good recommender system can recommend users the proper products, increase the user engagement, create more personalized pages for different users. There are several kinds of recommendation systems, including popularity based systems, Classification model based, etc. Assuming that individuals prefer things that are similar to other things they like and things that other people with similar tastes also like, we use a collaborative filtering system in this project.

As mentioned before, to get rid of bias as possible and reduce noise, we combine "overall", which is the score in integer range one to five the buyer gives, with the data obtained by sentiment analysis and normalize the final results into float range one to five.

Collaborative filtering is a common and popular method applied in recommender system aiming to fill in the missing entries of a user-item association matrix and make use of previous item ratings made by similar-minded individuals to forecast how someone might rank an item. In our project, we apply the famous **Singular Value Decomposition (SVD)** algorithm. Set r'_{ui} as the SVD algorithm prediction:

$$r'_{ui} = \mu + b_u + b_i + q_i^T p_u$$

To estimate all unknown parameters, we minimize the following regularized squared error:

$$\sum_{r_{ui} \in R_{train}} (r_{ui} - r'_{ui})^2 + \lambda(b_i^2 + b_u^2 + \|q_i\|^2 + \|p_u\|^2)$$

The minimization is performed by a very straightforward stochastic gradient descent (details are omitted).[8]

We first apply the SVD model from Surprise library to have an overview of its performance by obtaining the Root Mean Squared Error (RMSE) value.

$$RMSE = \sqrt{\frac{1}{|R'|}} \sum_{r'_{ui} \in R'} (r_{ui} - r'_{ui})^2$$

Here are the main codes in Figure21.

Some result samples in Figure22. It can be seen that the RMSE value of our SVD model is 0.3887, which is rather low meaning that the SVD algorithm is reliable to be applied in the recommender system.

```
trainset, testset = train_test_split(data, test_size=0.2)
algo = SVD()
predictions = algo.fit(trainset).test(testset)
predictions[:5]
accuracy.rmse(predictions)
```

Figure 21: Codes of Surprise SVD Model

```
[Prediction(uid='A1XFNWGTPMG1AE', iid='B00005M987', r_ui=3.188393718832071, est=3.679866146187297, details={'was_impossible': False}),
 Prediction(uid='A2AOQGG1QKRWQ', iid='B00000S2OE', r_ui=3.9220458553791886, est=3.5402798762215345, details={'was_impossible': False}),
 Prediction(uid='AM4ETI24CFO6D', iid='B00027JYOE', r_ui=3.419833819241983, est=3.582071830431944, details={'was_impossible': False}),
 Prediction(uid='A17PMFGCDROM78', iid='B00019TKVW', r_ui=3.871608818342152, est=3.469822636178092, details={'was_impossible': False}),
 Prediction(uid='A06Y9HMEN1XS5', iid='B00065XJ52', r_ui=3.347454051249476, est=3.151809039098597, details={'was_impossible': False})]

RMSE: 0.3887
```

Figure 22: Samples of SVD Prediction Results

4.3 Recommender

Our formal recommender is an SVD model-based collaborative filtering system. We first build a sparse matrix as Figure23.

asin / reviewerID	B00000016W	B00000064G	B000000OUJ	B000000TDH	B0000013GH	B0000013GT	B0000163G	B000001A5X	B000001A6N	B000001A6X	...
A10201L7BWW9RAX	0.0	0.0	0.0	0.0	0.0	0.000000	0.0	0.000000	0.0	0.000000	...
A10323WWTFPSQP	0.0	0.0	0.0	0.0	0.0	0.000000	0.0	0.000000	0.0	0.000000	...
A103KNDW8GN92L	0.0	0.0	0.0	0.0	0.0	0.000000	0.0	0.000000	0.0	0.000000	...
A103W7ZPKGOCC9	0.0	0.0	0.0	0.0	0.0	0.000000	0.0	3.800903	0.0	3.937612	...
A105S56ODHGJEK	0.0	0.0	0.0	0.0	0.0	3.628674	0.0	0.000000	0.0	0.000000	...

Figure 23: Pivot Table

Fill in the unknown values in the utility matrix with 0, because it is sparse as predicted, and then transport. After decomposing the matrix with TruncatedSVD to deal with the sparse matrix and correlate obtained result, based on things rated by other buyers who purchased the same product, this buyer's purchase and all other goods have a correlation, and giving a product ID, we can gain a list of top highly correlated products whose correlation is larger than a threshold. Main codes are as follows:

```
ratings_matrix = new_df.pivot_table(values='unbias_overall',
        index='reviewerID', columns='asin', fill_value=0)
X = ratings_matrix.T
SVD = TruncatedSVD(n_components=10)
decomposed_matrix = SVD.fit_transform(X)
correlation_matrix = np.corrcoef(decomposed_matrix)
```

Figure 24: SVD Model-Based Collaborative Filtering System

For example, given a product with ID "B005QJZ5FA" purchased by a customer, after removing products he has bought, we can get a list of recommended products arrayed by correlation degree in Figure25.

In conclusion, with the help of sentiment analysis on review texts, we reduce the noise and get rid of biases as possible in the raw data, and we have developed a recommender based on the SVD algorithm which is famous and popular in collaborative filtering techniques.

```
Recommend = list(X.index[correlation_product_ID > 0.65])

Recommend.remove(i)

Recommend[0:10]
```

```
['B000000TDH',
 'B000001DZO',
 'B00000J5WQ',
 'B000002J2S',
 'B000002VN7',
 'B000002VT6',
 'B00000DGUG',
 'B00000IAU3',
 'B00001QGQI',
 'B00004Z41Q']
```

Figure 25: Recommended Products

5 CONTRIBUTION TABLE

section	contributor
Dataset Analysis	Li Kaiyu
Simple search engine	Chang Lo-wei, Chen Yueqi
Review summarizer	Li Jiayi, Chen Lei
Application	Li Jiayi, Chen Lei

REFERENCES

[1] Krati Agarwal. 2020. *RAKE: Rapid Automatic Keyword Extraction Algorithm.* https://medium.datadriveninvestor.com/rake-rapid-automatic-keyword-extraction-algorithm-f4ec17b2886c

[2] Saurav Anand. 2020. *Recommender System Using Amazon Reviews.* https://www.kaggle.com/code/saurav9786/recommender-system-using-amazon-reviews

[3] Edward Loper Bird, Steven and Ewan Klein. 2009. *Natural Language Processing with Python.* O'Reilly Media Inc.

[4] Florian Boudin. 2016. pke: an open source python-based keyphrase extraction toolkit. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: System Demonstrations.* Osaka, Japan, 69–73. http://aclweb.org/anthology/C16-2015

[5] Jonathan Eyal, Wahyudi Soeriaatmadja, Grace Ho, and Denise Chong. 2022. Container Pages. https://www.straitstimes.com/

[6] Ruining He and Julian McAuley. 2016. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *proceedings of the 25th international conference on world wide web.* 507–517.

[7] Matthew Honnibal and Ines Montani. 2017. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. (2017). To appear.

[8] Nicolas Hug. 2015. *Matrix Factorization-based algorithms.* https://surprise.readthedocs.io/en/stable/matrix_factorization.html#unbiased-note

[9] Nicolas Hug. 2020. Surprise: A Python library for recommender systems. *Journal of Open Source Software* 5, 52 (2020), 2174. https://doi.org/10.21105/joss.02174

[10] Steven Loria. 2018. textblob Documentation. *Release 0.15* 2 (2018).

[11] Layla Oesper, Daniele Merico, Ruth Isserlin, and Gary D Bader. 2011. WordCloud: a Cytoscape plugin to create a visual semantic summary of networks. *Source code for biology and medicine* 6, 1 (2011), 7.

[12] Anusha Pai. 2014. Text Summarizer Using Abstractive and Extractive Method. *International Journal of Engineering Research Technology (IJERT)* 5 (May 2014). http://doi.acm.org/10.1145/1057270.1057278

[13] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[14] Pauli Virtanen and Gommers. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2