Web Information Retrieval – Final Project

This project was built during the course Web Information Retrieval, taught by Professor Sara Cohen In the Hebrew University of Jerusalem.

The project is a limited and simplified search engine, which enables the user to enter queries (including wildcards) about data sets of product reviews form the Stanford website.

During the data parsing we've treated each word as a token.

In order to determine, for each query, which reviews \ products id (depends on the query's type) the program constructs two types of data structures:

1. **Dictionary - remains on the main memory:**

   It is called a dictionary because it enables a quick search of a given token (the key) and the retrieving of a pointer to its inverted list on the disk (the value).

   In short:

   Each one of them implements the K-1 in K Front Coding, when K=10. Each one of them contains a concatenated string and a table which contains the following fields:

   - Frequency – int (4 bytes)
   - Posting Ptr – long (8 bytes)
   - Length – byte
   - Prefix Size – byte
   - Term Ptr – int (4 bytes)

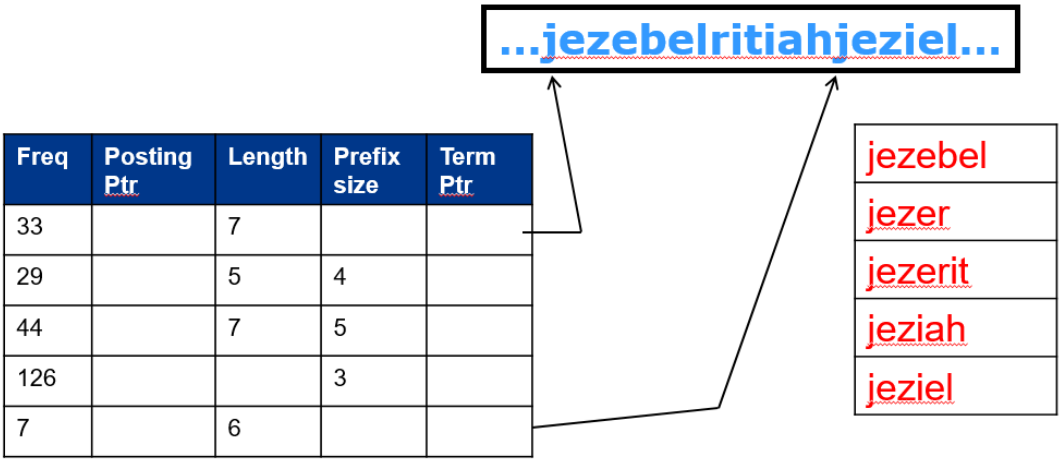
   Explanation:

   We've constructed two different dictionaries, one for the tokens and one for the product ids.

   Instead of storing each token \ id in a dictionary we've:

   a. Created one big string which is a "smart" and efficient concatenation of all the tokens \ ids.
   b. Stored in the dictionary only pointers to the token \ id in the concatenated string.

We say the concatenated string is efficient because it uses a method called “front coding” – it relies on the fact that adjacent sorted tokens have common prefixes. So, for each block of 10 consecutive tokens in the dictionary, we wrote in the concatenated string only once their common prefix in order to save space.
In the dictionary, every first token in a block is contained fully in the concatenated string, so we only stored a pointer to its start index. For each of the remaining 9 in the block, we stored its length, the length of its common prefix with the previous word and a pointer to the start of its unique part in the string – all this information is enough to construct the words quickly.



The division to blocks enables binary search in the dictionary and thus is important.

2. **<u>Inverted lists – a file containing posting lists for each token and id – stored on disk:</u>**

   We've constructed two different inverted indices, one for the tokens and one for the product ids. Each one of them implements Length-Precoded Varint coding (i.e., when encoding a number, the first two bits are used to indicate the number of bytes needed for the number’s encoding).

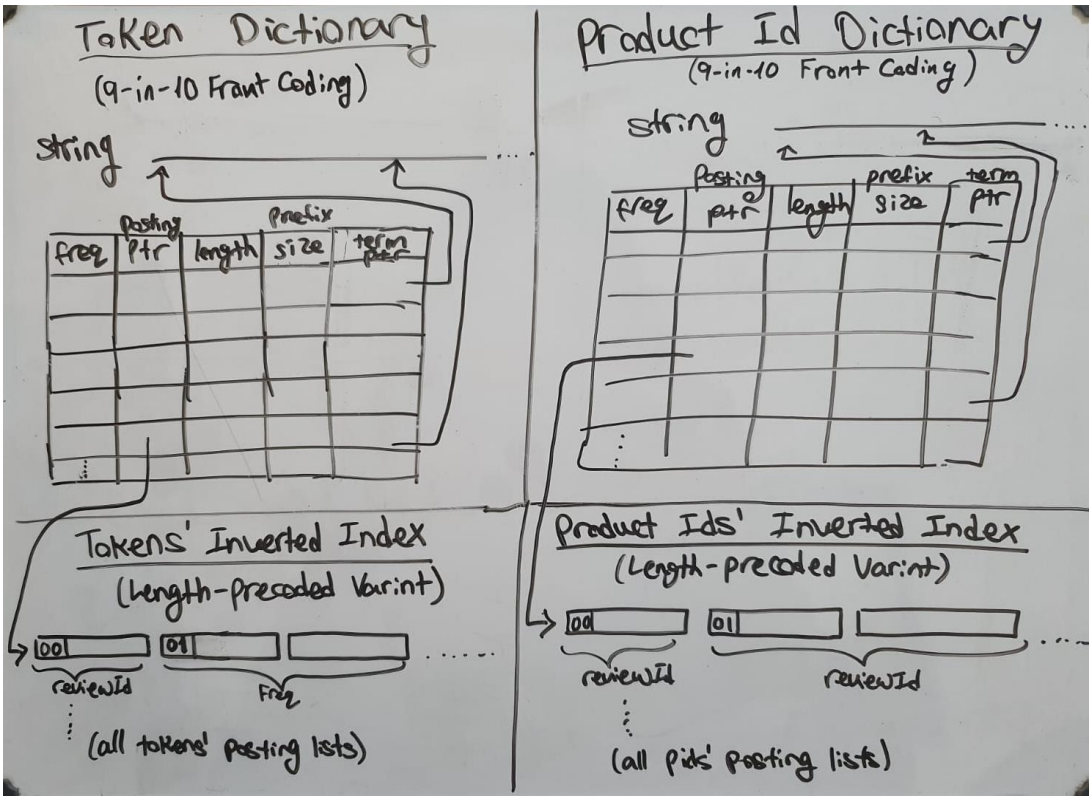
   - **The tokens' inverted index** contains a sequence of posting lists. Each posting list contains a sequence of reviewId (to be precise – the gaps between consequent ids) and frequency pairs (i.e. reviewId-1, freq-1, …, reviewId-n, freq-n where each reviewId is an id of a review in which the

term appears and freq is the number of its appearances in the review). Each number is coded in Length-Precoded Varint, with the required number of bytes.

- **The product ids' inverted index** contains a sequence of posting lists. Each posting list contains a sequence of (gaps between) reviewIds (i.e. reviewId-1, …, reviewId-n). Each number is coded in Length-Precoded Varint, with the required number of bytes.

00000001 00001111 01000011 11111111 10000111 11111111 11111111

1 15 511 131071



We built the indexes in an efficient way – using sort-merge technique.

Answering Queries

For a given query, we wanted rank each review \ product id (depends on the query's type) according to how much if fits the query, and return the most fitting ones by order.

We've used the information int the dictionaries and the posting lists in order to implement 2 different searches, each of them is based on a ranking method we saw in class:

1. Vector Space Model search – with the method vectorSpaceSearch in the class ReviewSearch.
2. Language Model ranking search - with the method languageModelSearch in the class ReviewSearch.

The two searching methods above return the most fitting reviews.

Another option our code provides is searching for products instead of reviews; The method productSearch returns the most fitting products by order, and works this way:

ProductSearch function description

Let's call the input query $\boldsymbol{q}$.

1. In this function we first ranked all the reviews which contain at least one word from $\boldsymbol{q}$, using the same rank logic we used in vectorSpaceSearch function, calling this rank $\boldsymbol{rank(r)}$, when $\boldsymbol{r}$ is the reviewId.

   Let the productIds of all the results of this stage be $\boldsymbol{P}$ and let $\boldsymbol{R(p)}$ be all the reviews of the productId $\boldsymbol{p} \in \boldsymbol{P}$.
2. In the next step, we built a hashMap in which a key is $\boldsymbol{p} \in \boldsymbol{P}$ and a value is a weighted mean of all the ranks of $\boldsymbol{R(p)}$.

   So, for all $p \in P$, The weighted mean looks like:

   $$\frac{1}{|R(p)|}\sum_{r \in R(p)}\left(\lambda \cdot \left(rank(r) \cdot \frac{score}{5}\right) + (1-\lambda) \cdot \left(help \cdot rank(r)\right)\right)$$

   When $\lambda = 0.8$.
3. We will return the k product ids with the highest rank.

In this way, we considered:

- the reviews rank of each product id
- the score of each review of the product id
- the helpfulness parameter of each review of the product id
- because we took the mean of each product id's reviews, the number of reviews does not effect the rank of this product.

את החלק לעיל ביצענו במסגרת 3 תרגילים של הקורס, החלק להלן הוא הפרוייקט הסופי שלנו:

פרויקט סופי בקורס "אחזור מידע באינטרנט" 67782

מגישים: מור משה ועדו פורת | קישור לסרטון של הפרויקט

נושא הפרויקט:

בחרנו בפרויקט מעשי בו הרחבנו את תוצר התרגילים שמימשנו בקורס כך שיתמוך בהרצת שאילתות עם wildcards. מימשנו את התמיכה הזו באמצעות שני מנגנונים שונים עליהם למדנו במהלך השיעורים, bigram-index ו-rotated lexicon, ולאחר מכן השווינו את הביצועים שלהם על מאגרי מידע שונים. ההשוואה שביצענו בחנה מספר היבטים: גודל האינדקס, זמן יצירת האינדקס, זמן הרצת שאילתה ועוד.

מבוא:

בהינתן token המכיל את הסימן המיוחד *, המטרה שלנו היא להחזיר את כל התוצאות של כל ה-token-ים המתאימים למילת השאילתה. כפי שלמדנו בהרצאה, נעשה זאת ע"י אחזקת שני Index-ים:

- אינדקס עזר שבאמצעותו נמצא את כל ה-tokens שמתאימים לתבנית מילת השאילתה.
- אינדקס ראשי, באמצעותו נחפש את המסמכים המתאימים לכל ה-tokens שמצאנו בשלב הקודם.

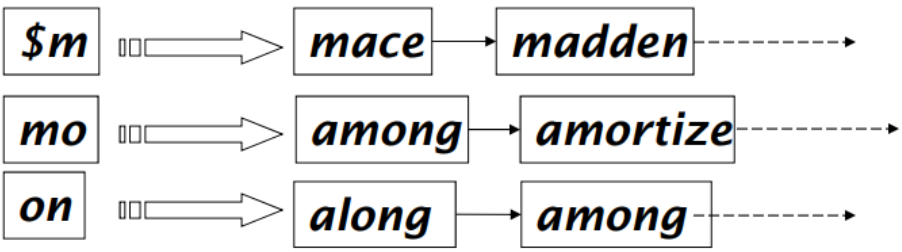
בחנו, כאמור, שני סוגים של אינדקסי עזר:

1. **N-gram Index:**

נעבור על כל ה-tokens במאגר, עבור כל token נוסיף סימן ייחודי של $ בתחילתו ובסופו ונפצל את התוצאה לרצפים בגודל n. את כל הרצפים שנקבל נסדר לפי סדר אלפביתי במבנה נתונים כלשהו, כאשר כל רצף מצביע לרשימת כל ה-tokens שמכילים אותו.

אנחנו בחרנו לממש n-gram index עם n=2, כלומר bigram index.

להלן דוגמה לאינדקס שכזה:



בהינתן token עם כוכבית אחת או יותר:

- נוסיף את התו הייחודי $ בתחילת ה-token ובסופו.
- נפצל אותו לרצפים שלא מכילים כוכבית.
- נחלק כל רצף שכזה לתת-רצפים באורך 2 (bigrams).
- עבור כל תת-רצף מהסעיף הקודם, נשלוף מה-bigram index את רשימת ה-tokens המתאימים לו.
- נבצע חיתוך בין כל הרשימות שקיבלנו בסעיף הקודם ונחזיר את התוצאה.

2. **Rotated Lexicon Index:**

ניזכר כי i-rotation של token אותו נייצג כ - $t = c_1, c_2, \dots, c_n$ הינו $c_i, \dots, c_n, c_1, \dots, c_{i-1}$. כמו כן, נשתמש בסימון (k, i) עבור הסיבוב ה-i של ה-token במקום ה-k באינדקס הראשי (ספציפית ב-dictionary).

על מנת לסמן את תחילתו של כל token נוסיף לפני האות הראשונה את התו הייחודי $.

נשים לב כי כפי שלמדנו בהרצאה, (k, i) מזהה באופן ייחודי סיבוב של token.

ה-rotated index יכיל את כל הסיבובים האפשריים של כל ה-tokens במאגר, כאשר כל סיבוב מסומן על ידי (k, i) ואלו ממוינים בסדר אלפביתי.

Address
(1,1)
(2,1)
(3,1)
(4,1)
(1,2)
(3,3)
(4,3)
(1,6)
(2,5)
(3,6)
(4,7)

ה-rotatrd lexicon

Term Number	Term
1	abhor
2	bear
3	labor
4	labour

ה-dictionary

נציג כאן את המימוש ההתחלתי שלנו ובהמשך נציג את השינויים שביצענו וההשפעות שלהם על ביצועי התוכנה. בהינתן token עם כוכבית אחת או יותר:

- נוסיף את התו הייחודי $ לתחילת ה-token (ורק לתחילתו).

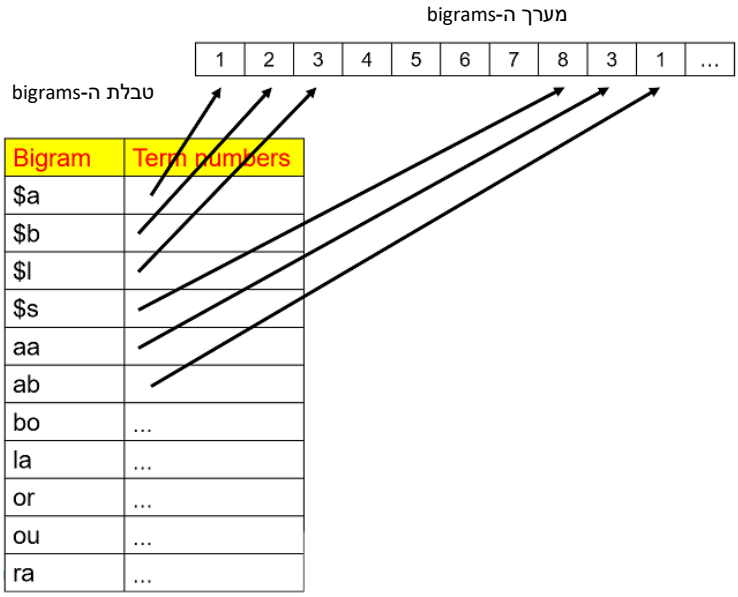
- נסובב את ה-token החדש עד שהכוכבית הראשונה תהיה בסוף ה-token[1]. המטרה מאחורי המהלך הזה היא לקבל פחות רצפים וכתוצאה מכך לבצע כמה שפחות חיפושים ב-index העזר. למשל, עבור $inte*net, במקום שנצטרך לחפש גם את $inte וגם את net, נחפש רק את net$inte.
- נפצל את התוצאה מהסעיף הקודם לרצפים שלא מכילים כוכבית.
- עבור כל רצף שכזה, אנחנו רוצים למצוא את כל ה-rotations ב-rotated index שמתחילים באותו הרצף (למשל באמצעות חיפוש בינארי).

  מכיוון שיכולות להיות כמה תוצאות מתאימות (כי יכולים להיות הרבה rotations של tokens במאגר עם התחילית הרלוונטית), לאחר שמצאנו ערך מתאים נצטרך לחפש בסביבתו את כל הערכים המתאימים הנוספים (אם יש כאלו). נשמור את ערכי ה-k של כל התוצאות (הן מהצורה (k, i)).
- נחשב את החיתוך בין התוצאות שקיבלנו עבור כל רצף מהסעיף הקודם ונחזיר את התוצאה, שהיא ה-id-ים של כל ה-tokens המתאימים.

מימוש:

1. **Bigram-index:**

ראשית, על מנת לחסוך במקום, נשמור את ה-id-ים של ה-tokens, ולא את ה-tokens עצמם. גם עבור ה-bigrams נשמור את ה-ids שלהם (לפי סדר אלפביתי) ולא את ה-string שלהם. בנוסף, במקום לשמור עבור כל bigram רשימה משל עצמו, נשמור את כל הרשימות בצורה רציפה במערך אחד, כאשר לכל bigram באינדקס העזר יהיה מצביע למקום במערך בו הרשימה שלו מתחילה (נדע כמה מספרים לקרוא מהמערך לפי המצביע של ה-bigram הבא אחריו). להלן דוגמא:



[1] קיבלנו את הרעיון מהספר Managing Gigabytes

נתייחס למבנה הנתונים המכיל את ה-bigrams ואת המצביעים כ"טבלת ה-bigrams" ולמערך ה-token ids בתור "מערך ה-bigrams".

<u>כתיבה לדיסק:</u>

כאשר כתבנו את ה-bigram index לדיסק, חילקנו אותו לשני קבצים – האחד עבור הטבלה והשני עבור המערך:

- את המערך שמרנו בקובץ בשם bigramIndex. כדי לחסוך במקום, עבור כל רשימת token ids של bigram, שמרנו את ההפרשים בין ה-ids וכן השתמשנו בקידוד Length-Precoded Varint coding.
- את הטבלה שמרנו בקובץ בשם bigramPointers. לכל Bigram נתנו מספר ייחודי לפי הסדר האלפביתי שלהם ושמרנו בקובץ, עבור כל bigram, את המספר הייחודי שלו (bite אחד, שכן יש בדיוק $37 \cdot 37$ כאלו) ואת המצביע שלו למערך (long).

<u>קריאה מהדיסק:</u>

משום שטבלת ה-bigrams היא בגודל קבוע ($37 \cdot 37$ משום שיש 37 תווים שונים), נשמור אותה בזיכרון המרכזי וכך נאפשר גישה יעילה.

עם זאת, גודל מערך ה-bigrams אינו חסום מלעיל ותלוי במספר המילים השונות במאגר (כפי שראינו בהרצאה גודלו הוא $\frac{\lceil \log w \rceil}{8} \cdot w(n+1)$). לכן, בהתחלה בחרנו שלא לקרוא אותו לזיכרון המרכזי וקראנו ממנו לפי הצורך על מנת לשמור על scalability. עבור כל bigram שמרנו offset בקובץ שבדיסק והשתמשנו ב-randomAccessFile כדי לגשת ישירות לרשימת ה-token ids של אותו ה-bigram.

כאשר סיימנו לממש את הפרויקט, החלטנו לנסות לקרוא את כל המערך לזיכרון המרכזי וראינו כי הצלחנו לעשות זאת גם עבור קבצים גדולים. על כן, ומשום שראינו כי זמני הריצה השתפרו, החלטנו לבסוף לקרוא את המערך לזיכרון המרכזי.

<u>יצירת האינדקס:</u>

בעת קריאת קובץ הקלט, כאשר עיבדנו token מסוים, שמרנו במערך עבור כל bigram שנמצא באותו ה-token את ה-pair הבא: (bigram id, token id), כאשר ה-id של כל חלק ב-pair נקבע לפי הסדר האלפביתי. לבסוף, קיבלנו רשימת pairs מהצורה הנ"ל, מיינו אותה, ועבור כל bigram id, חילצנו את רשימת ה-token ids שמכילים אותו. ביצענו זאת באמצעות האלגוריתם sort-merge. בכל שלב, כאשר המערך התמלא, כתבנו אותו לקובץ שונה בדיסק, לאחר שמיינו אותו. כשסיימנו

לעבד את הקובץ, התחלנו בתהליך ה-merge, עד שלבסוף נשארנו עם קובץ אחד שממנו חילצנו את האינדקס בצורה יעילה.

<u>הערות</u>:

- בהינתן token שמכיל wildcards, לאחר שקיבלנו את התוצאות של ה-tokens המתאימים לו כפי שהסברנו בחלק המבוא, נאלצנו להתמודד שתי בעיות:

  א. נניח שהקלט שלנו הוא *mon, אזי ה-bigrams שנחפש יהיו: $m, mo, on.
  כאשר נבצע חיתוך של התוצאות עבור כל bigram, אנו עלולים לקבל גם את המילה moon, על אף שהיא לא מתאימה (כיוון שהיא אכן מכילה את כל ה-bigrams המצוינים לעיל)[2].

  ב. נניח שהקלט שלנו הוא *ab*r, כאשר נפצל אותו לרצפים שלא מכילים כוכבית, נקבל בין השאר את הרצף r, שהוא אינו bigram. איך נתמודד עם זה?

  <u>פתרון</u>: לאחר הפיצול לרצפים ללא כוכבית, נתעלם בשלב זה מכל החלקים שאורכם תו אחד. לאחר שנקבל את תוצאת החיתוך, עבור כל token שנמצא בה, נבצע post-filtering, כלומר נוודא שהוא מתאים ל-regex של הקלט. כך למעשה נסנן מילים כמו moon, וגם נוודא שהתוצאות מכילות את התווים שהורדנו בסינון של מקרה ב'.

- נשים לב, כי נותרנו עם מקרה הקצה הבא:
  נניח שקיבלנו קלט מהצורה *a*. קלט זה אינו מכיל שום bigram. על מנת להתמודד עם מקרה זה, לקחנו את כל הזוגות האפשריים של bigrams שמכילים את התו a, וביצענו עבורם את החיפוש הרגיל. כך למעשה ניצלנו את ה-bigram index, ונמנענו ממעבר על כל המילים במאגר.
- מקרה קצה נוסף הוא הקלט *. במקרה זה, החזרנו את כל התוצאות במאגר.
- בשיעור למדנו כי בהינתן Bigram, כדי לחלק את רשימת ה-tokens ids שלו, אנחנו נאלצים לבצע חיפוש בינארי. אולם, כיוון שגודל ה-bigrams הוא סופי, בחרנו לשמור את כל ה-bigrams הקיימים באינדקס (גם כאלו שלא הופיעו כלל במאגר, ואז הרשימה שלהם תהיה בגודל 0). כך, לכל bigram יש אינדקס ייחודי (לפי המיקום שלו בסדר האלפביתי) ובאופן זה לא נצטרך לבצע חיפוש בינארי, אלא ניגש לאינדקס של ה-bigram במערך. כך, למעשה, שיפרנו את זמן ריצת החיפוש להיות O(1).

2. <u>**Rotated Lexicon Index:**</u>

[2] קראנו על הבעיה הזו והפתרון שלה <u>במצגת מהאתר של Stanford</u>, שקף 19

עבור כל token במאגר, שמרנו את כל הסיבובים האפשריים שלו. כל איבר באינדקס יהיה מהצורה (k,i) כאשר k הוא מספר ה-token (int), ו-i הוא הסיבוב (byte). נניח כי במאגר יש w מילים ייחודיות, וכי האורך הממוצע של מילה הוא n, אז מספר הרשומות יהיה $w(n+1)$.

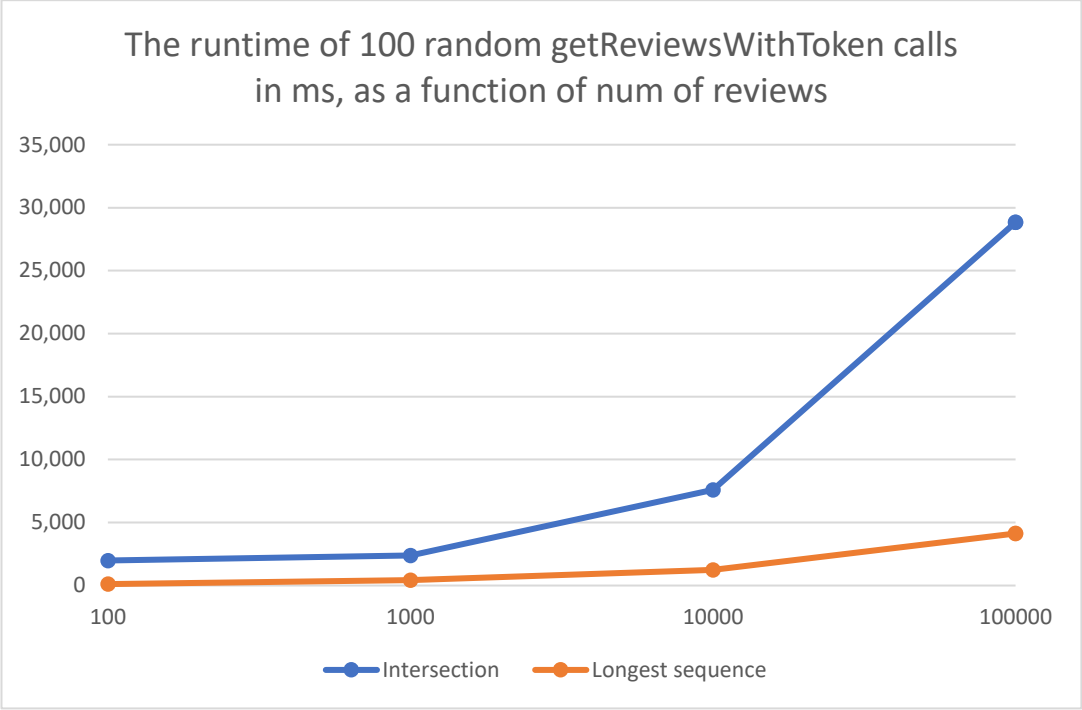
<u>כתיבה לדיסק:</u> כתבנו את האינדקס כולו לקובץ אחד בשם *rotatedLexiconFile*, כפי שהסברנו לעיל. גודל הקובץ יהיה: $w(n+1) \cdot 5$.

<u>קריאה מהדיסק:</u> בחרנו לקרוא את האינדקס הזה לזיכרון המרכזי בשלמותו.

<u>יצירת האינדקס:</u> בזמן עיבוד קובץ הקלט, עבור כל *token* הוספנו את התו הייחודי $ להתחלה ושמרנו את כל הסיבובים של התוצאה באינדקס.

<u>הערה:</u> כאשר קיבלנו *token* עם *wildcards*, לאחר שפיצלנו אותו לרצפים ללא *wildcard*, במקום לבצע חיפוש בינארי ב-*rotated lexicon index* עבור כל רצף ולבסוף לבצע ביניהם חיתוך, לקחנו את הרצף הארוך ביותר ורק עליו ביצענו את החיפוש וקיבלנו את ה-*tokens ids* המתאימים לו. לאחר מכן ביצענו *post-filtering* כדי לסנן מילים שלא תואמות לשאר הרצפים שלא חיפשנו לפיהם. בצורה זו, הצלחנו לשפר משמעותית את זמני הריצה של חיפוש מילים עם *wildcards*[3]. להלן דוגמה לשיפור הניכר בזמני הריצה:

[3] את הרעיון לקחנו מהספר Managing Gigabytes


The runtime of 100 random getReviewsWithToken calls in ms, as a function of num of reviews
35,000
30,000
25,000
20,000
15,000
10,000
5,000
0
100
1000
10000
100000
Intersection
Longest sequence

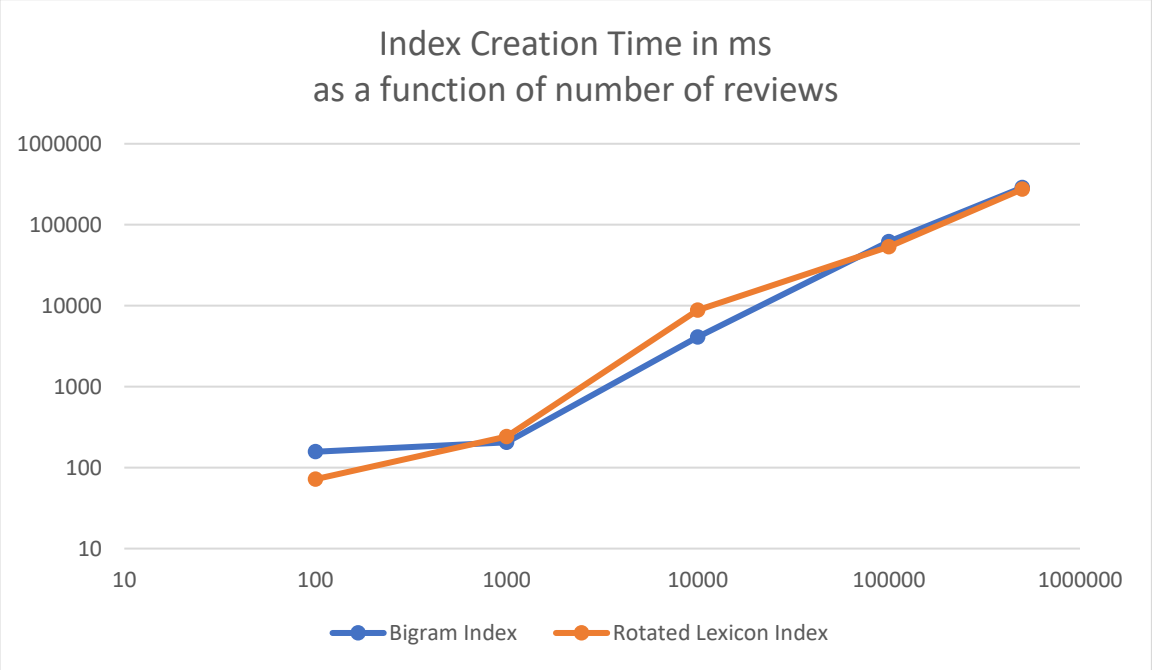
ניסויים:

ביצענו השוואות בין זמני הריצה וגדלי האינדקס, בין שני סוגי האינדקס השונים.

את המדידות ביצענו על קבצים בגדלים הבאים:

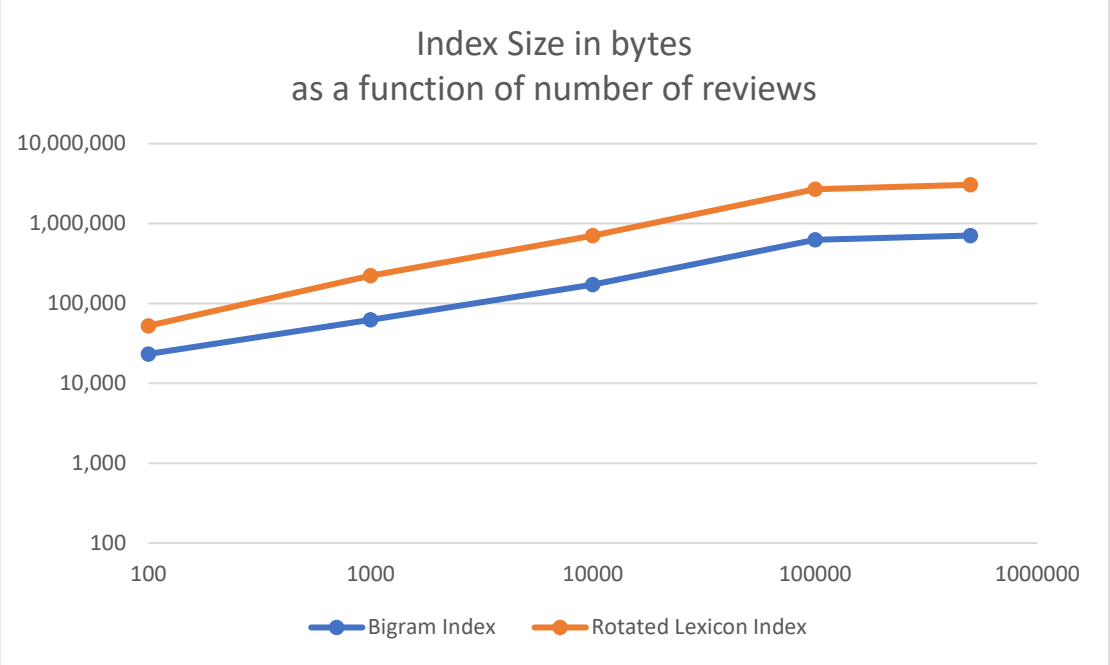
1. 100 (הקובץ שסופק לנו במודל בתרגיל הראשון)
2. 1,000 (הקובץ שסופק לנו במודל בתרגיל הראשון)
3. 10,000 (כשליש ממאגר הקובץ *Arts.txt* מהאתר של סטנפורד)
4. 100,000 (הקובץ *Home_&_Kitchen.txt* מהאתר של סטנפורד)
5. 500,000 (הקובץ *Clothing_&_Accessories.txt* מהאתר של סטנפורד)

להלן התוצאות:



The data of the graph above

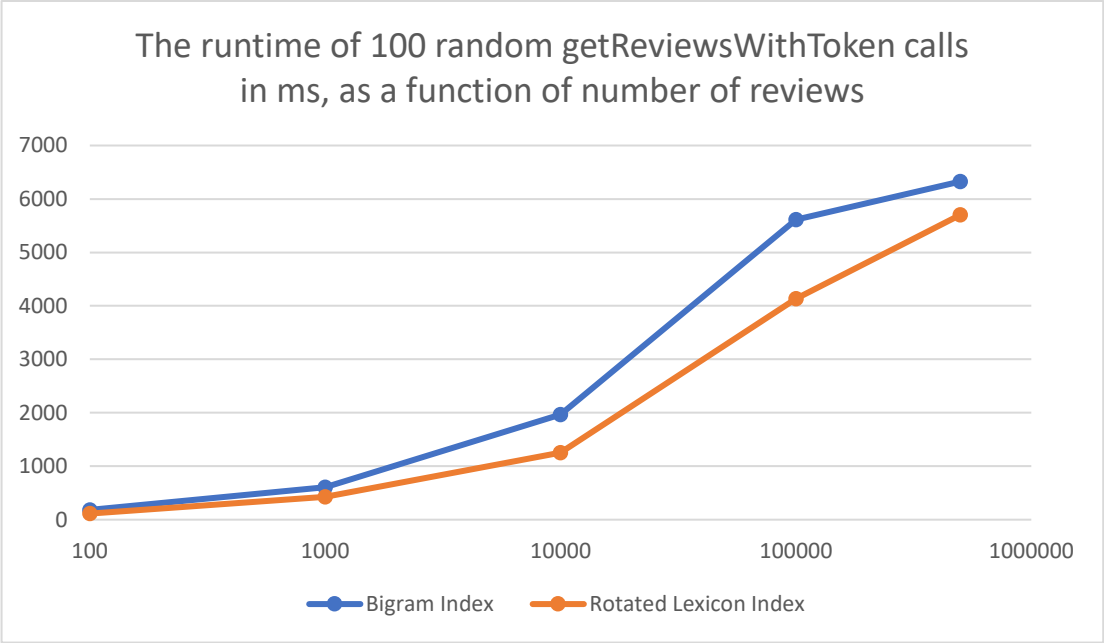
	100	1,000	10,000	100,000	500,000
Bigram Index	157	206	4,097	61,747	287,774
Rotated Lexicon Index	72	241	8,796	53,505	275,181



- מדדנו רק את החלק של ה- bigram/rotated lexicon ולא את האינדקס כולו.

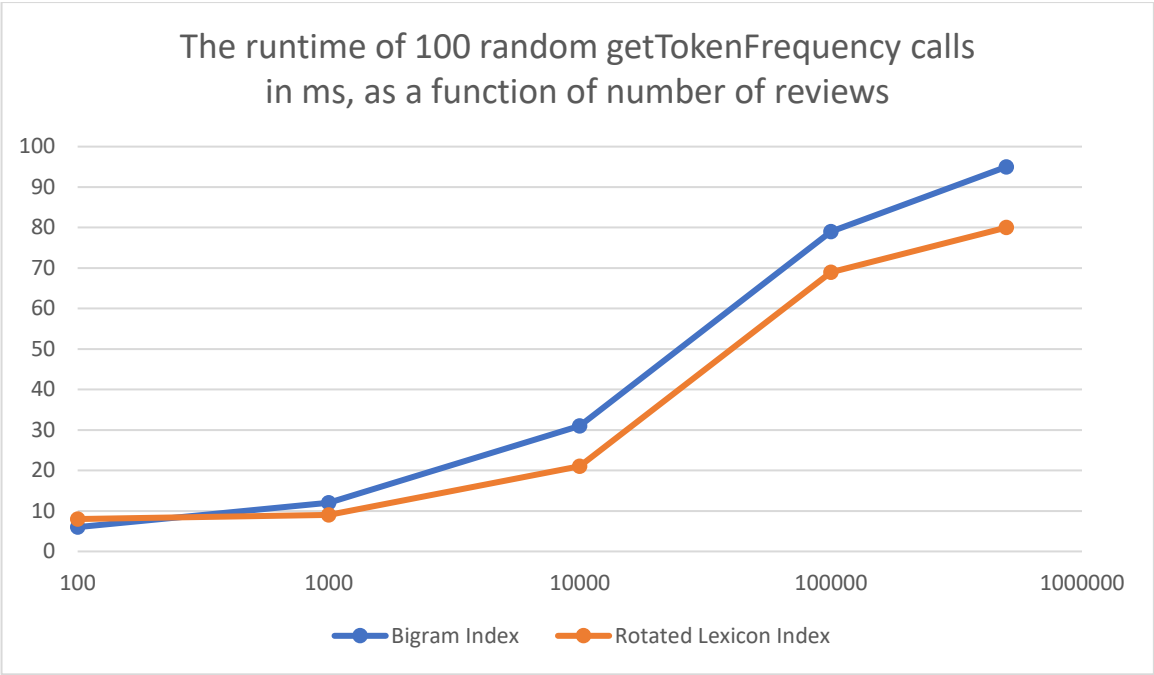
The data of the graph above

	100	1,000	10,000	100,000	500,000
Bigram Index	23,284	62,456	171,276	625,201	707,046
Rotated Lexicon Index	52,459	223,034	701,954	2,690,654	3,053,579



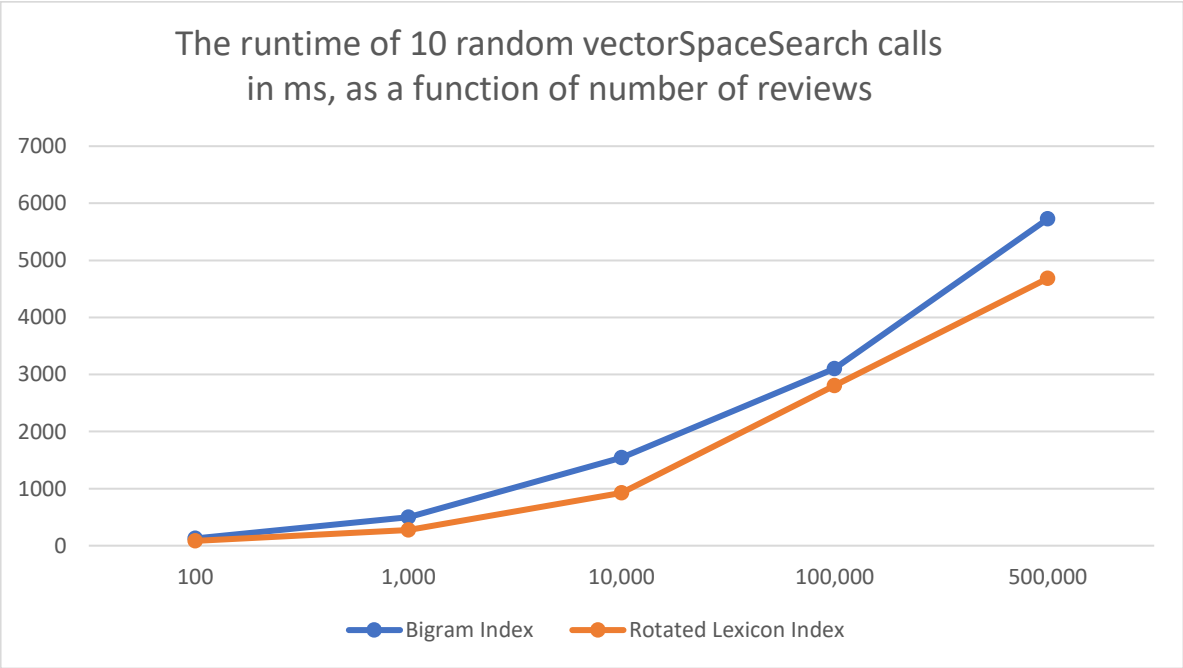
The data of the graph above

	100	1,000	10,000	100,000	500,000
Bigram Index	183	606	1,968	5,615	6,327
Rotated Lexicon Index	114	430	1,253	4,132	5,704



The data of the graph above

	100	1,000	10,000	100,000	500,000
Bigram Index	6	12	31	79	95
Rotated Lexicon Index	8	9	21	68	80



The data of the graph above

	100	1,000	10,000	100,000	500,000
Bigram Index	127	498	1,541	3,104	5,725
Rotated Lexicon Index	84	276	927	2,802	4,683

<u>מסקנות:</u>

1. **זמן יצירת האינדקס:**

זמני הריצה יצאו דומים, אין הבדלים משמעותיים בין הסוגים. כשאנחנו יוצרים את ה-*bigram index*, אנחנו עוברים על כל ה-*tokens* במאגר, ומפצלים כל אחד לזוגות (*bigrams*), לכן זמן הריצה יהיה $O(w \cdot n)$ כאשר *w* הוא מספר ה-*tokens* השונים במאגר, ו-*n* הינו האורך הממוצע של *token*. גם ב-*rotated lexicon*, אנחנו עוברים על כל ה-*tokens* במאגר, ועבור כל אחד יוצרים את כל הסיבובים שלו, ולכן גם פה זמן הריצה יהיה $O(w \cdot n)$. כלומר, בשני המקרים, זמן הריצה יהיה זהה פחות או יותר.

2. **גודל האינדקס:**

כפי שציפינו, ניתן לראות כי תמיד גודל האינדקס של *rotated lexicon* יהיה גדול משל ה-*bigram*. כמו כן, היחס בין הגדלים נשאר דומה ככל שכמות הביקורות עולה (בערך פי 4). ככל שהקורפוס גדול יותר, המחיר שנשלם עבור גודל ה-*rotated lexicon index* יהיה יקר יותר משמעותית (ההפרש בגדלים הוא אקספוננציאלי). לכן אם ב-*tradeoff* בין מקום לזמן ריצה יותר חשוב לנו המקום, אז עבור קורפוסים גדולים נעדיף להשתמש ב-*bigram index*.

3. **זמן ריצת 100 קריאות של *getReviewsWithToken*:**

ראשית, קיבלנו כי זמני הריצה של *rotated lexicon index* טובים יותר.
אולם, ציפינו כי ההבדלים יהיו משמעותיים ועקביים יותר בהשוואה לתוצאות שקיבלנו.
ניסינו להבין מדוע התוצאות שקיבלנו לא תואמות את הציפיות שלנו, והגענו למסקנה הבאה: בחיפוש ב-*rotated lexicon* אנחנו מבצעים חיפוש בינארי על הטבלה, ומעבר על כל "הסביבה" כדי למצוא את כל התוצאות המתאימות. לעומת זאת, ב-*bigram index*, כל חיפוש של *bigram* מתבצע ב-*O(1)*. באופן זה, למרות שב-*bigram* אנחנו מבצעים יותר חיפושים ועליהם מבצעים חיתוך, בסופו של דבר החיפוש המהיר "מתקזז" ולכן ההפרשים בזמני הריצה לא גדולים משמעותית.

4. **זמן ריצת 100 קריאות של *getTokenFrequency* ו-10 קריאות של *vectorSpaceSearch*:**

באופן דומה לסעיף הקודם, קיבלנו כי זמני הריצה של *rotated lexicon index* טובים יותר, אך לא באופן משמעותי כפי שציפינו. זאת מאותה סיבה שהסברנו קודם.

<u>לסיכום:</u>

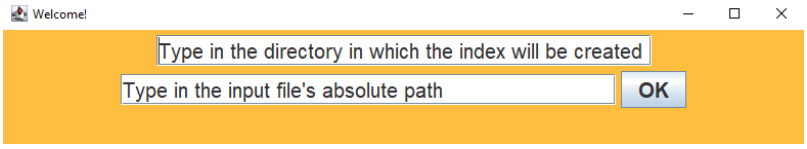
אם אנחנו מוגבלים במקום בדיסק, נבחר להשתמש ב-*bigram index* כאשר לא נשלם מחיר משמעותי בזמני הריצה. מצד שני, אם אין לנו בעיית זיכרון והמטרה שלנו היא זמני ריצה מהירים ככל האפשר, נבחר להשתמש ב-*rotated lexicon*.

הוראות הרצה:

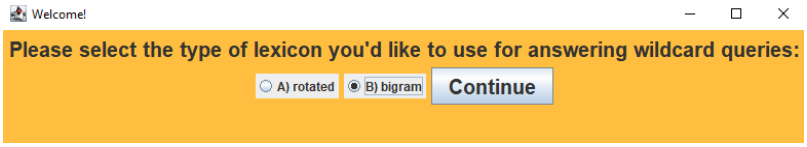
ישנן שתי אפשרויות להריץ את התוכנית שכתבנו:

1. בדיוק כמו שמריצים את תרגילי הקורס (כדי להפעיל את התוכנית כך יש למחוק את הפונקציה *main*). יש לשים לב שבחלק מהחתימות יש לשלוח את הפרמטר *isRotated*, המציין האם על התוכנית להשתמש ב-*rotated lexicon* או לא (כלומר, להשתמש ב-*bigram index*).
2. פשוט להריץ את התוכנית ללא שום ארגומנטים ולפעול אחר השלבים הבאים:

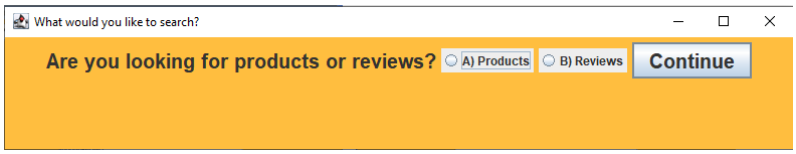
א. להכניס כקלט את הכתובות המלאות של התיקייה בה ייכתב האינדקס ושל קובץ הקלט:



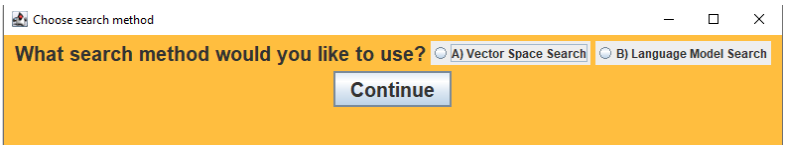
ב. יש לבחור באיזה אינדקס להשתמש – *bigram* או *rotated*:



ג. יש לבחור מהי מטרת החיפוש: מוצרים או חוות דעת. עבור חוות דעת יש לעבור לסעיף ד, עבור מוצרים – לסעיף האחרון.



ד. יש לבחור באיזה מודל חיפוש להשתמש: *Vector Space* או *Language Model*



ה. להזין שאילתה (המורכבת ממילה אחת או יותר), עם או בלי *wildcards*, ללחוץ *Search* ולהנות מהתוצאות!