אחזור מידע – פרויקט להגשה – אביב התשפ"ד

יש להגיש למוודל עד לתאריך 25.8.24 בשעה 23:59

*מבוסס על תרגילי הבית בקורס*

תרגיל בית 1

תרגיל בית 2

תרגיל בית 3

1. מהו תחום העיסוק המרכזי של האתר? מהו המידע הזמין למשתמשי האתר? ענו בפסקה אחת. צרפו את הקישור לאתר.
2. רשמו שלוש שאילתות מעניינות שהייתם רוצים לקבל עליהן תשובה באתר, והאתר אינו **עונה עליהן** כעת. עבור כל שאילתא כזו, ציינו מהם פרטי המידע הנדרשים לצורך מענה על השאילתא.
3. בנו זחלן המחזיר את התוצאות לשאילתות שהגדרתם. מומלץ (אך לא חובה) לבנות את הזחלן בפייתון. הציגו את קוד הזחלן.
4. רשמו טכנולוגיות שונות מעניינות שהשתמשתם בהם בפרויקט.
5. כמה זמן רצו השאילתות שלכם?במה זה תלוי? האם לדעתכם ניתן לשפר זמן זה?
6. בנו inverted index ל- 15 המילים הנפוצות שחזרו (מספיק להתיחס ל -20 הדפים הראשונים שחזרו)
7. בחרו את אחת השאילתות, וחשבו tfIdf של המושגים בשאילתא.
8. האם בדפים שהוחזרו קיימים hubs? Authorities? נמקו.
9. בחרו 10 דפים שהזחלן החזיר, ואשר יש ביניהם קישורים .חשבו pageRank לכל דף, מומלץ להעזר בסקריפט מוכן. הציגו את החישובים ואת הדירוג הסופי.
10. הראו לשני משתמשים שונים את הדירוג מהסעיף הקודם, בקשו מהם לסמן relevance feedback .

חשבו precision ו - recall.האם ניתן להציע שאילתא מותאמת על מנת לשפר תוצאות?

1. הציגו את כל התוצרים של סעיפים א-ו בדף HTML יחיד. הדף יכול להיות דינמי או סטטי (כלומר מציג צילומי מסך של התשובות לשאלות). העלו את הדף לgit pages. צרפו קישור לדף שהעליתם.
2. בתאריך 12.8 תציגו את הדף מסעיף 11, הכולל את סעיפים 1-10. תקבלו משוב מחבריכם. ענו בטבלה:

|  |  |
| --- | --- |
| איזה שינוי הוצע? | האם לדעתכם יש מקום לשיפור זה? אם כן - הסבירו כיצד ניתן לשפר. אם לא, נמקו |
|  |  |
|  |  |

1. מהו היו האתגרים בקורס / פרויקט כצוות?

יש להגיש קישור לריפו שלכם ב -GIT, הכולל את כל קבצי האתר, ותיקייה בשם project שבה קובץ וורד ובו מענה על השאלות.

ניתן לפנות אליי בכל שאלה.

בהצלחה!

**פרויקט אחזור מידע**

**מגישים:**

**שם הקבוצה : Rank Rangers**

שיר כהן, ת.ז. 316216340

אלמוג קדוש, ת.ז. 315699439

עמית וינוגרד, ת.ז. 316597723

אלמוג אלבז, ת.ז. 213037369

**קישור לGitPages :**

[**https://cohensh96.github.io/Rank-Rangers--Information-Retrieval/FinalProjectDisplay.html**](https://cohensh96.github.io/Rank-Rangers--Information-Retrieval/FinalProjectDisplay.html)

**קישור לGitHub:**

[**https://github.com/cohensh96/Rank-Rangers--Information-Retrieval**](https://github.com/cohensh96/Rank-Rangers--Information-Retrieval)

* + - 1. **מהו תחום העיסוק המרכזי של האתר? מהו המידע הזמין למשתמשי האתר? ענו בפסקה אחת. צרפו את הקישור לאתר.**

האתרBundesgesundheitsministerium   הוא האתר הרשמי של משרד הבריאות הפדרלי בגרמניה. האתר נועד לספק מידע נגיש ומדויק לציבור הרחב ולגורמים מקצועיים בתחום הבריאות. המוקד המרכזי של האתר הוא יצירת ממשק שקוף בין הממשל לציבור בכל הנוגע למערכת הבריאות הלאומית. באתר ניתן למצוא מידע נרחב הכולל מדיניות בריאות, יוזמות ממשלתיות, רפורמות ופרסומים רשמיים בנושאי תקציבים וחקיקה. האתר מספק גם הסברים מפורטים על תוכניות ביטוח בריאות ציבוריות ופרטיות, זכויות וחובות המטופלים, והדרכה כיצד לגשת לטיפולים רפואיים. בנוסף, האתר מציע עדכונים על מצבי חירום רפואיים כמו מגפת הקורונה, תוכניות חיסונים וסטטיסטיקות על מצב מערכת הבריאות והתשתיות בגרמניה. האתר כולל גם שירותים מקוונים כמו טפסים, מסמכים ושאלות נפוצות. האתר משמש כמשאב מרכזי לקידום שקיפות, העלאת מודעות בריאותית ומתן כלים לאזרחים, חוקרים ואנשי מקצוע רפואיים להבנת מערכת הבריאות בגרמניה.

כתובת האתר:

<https://www.bundesgesundheitsministerium.de/en/>

* + - 1. **רשמו שלוש שאילתות מעניינות שהייתם רוצים לקבל עליהן תשובה באתר, והאתר אינו עונה עליהן כעת. עבור כל שאילתא כזו, ציינו מהם פרטי המידע הנדרשים לצורך מענה על השאילתא.**

**Query 1**:

* “What are the available healthcare policies for chronic diseases in Germany?״

**Required Information**:

* Comprehensive list of government policies targeting chronic diseases, e.g., diabetes, heart disease, cancer.
* Details about specific programs or initiatives for prevention, diagnosis, and treatment.
* Eligibility criteria for accessing these programs.
* Budget allocation and funding for chronic disease management.
* Information about partnerships with private or non-governmental organizations

**Query 2**:

* “What are the average waiting times in hospitals in Germany by region?”

**Required Information**:

* A clear definition of the regions in Germany, federal states.
* Hospitals categorized by these regions.
* A list of hospitals in each region.
* Types of hospitals, public, private, university hospitals, specialized clinics.
* The types of services for which waiting times are measured, emergency services, elective surgeries, specialist consultations.
* Average waiting times for each service in each hospital.
* Patient demographic information, such as age groups or specific needs, pediatrics, geriatrics.
* Population density and healthcare demand in each region.
* Regional breakdown of these costs, e.g., by federal state or district.
* Regional differences in healthcare policies that may affect waiting times, funding, resource allocation, staffing levels.
* in implementing these policies.

**Query 3**:

* "How can I replace my lost European Health Insurance Card (EHIC)?"

**Required Information**:

* Who can apply for a replacement e.g. EU citizens, UK residents with agreements etc.
* Where to apply - online, by phone, in person.
* Required documents such as proof of identity, residency, previous EHIC details.
* Is there a fee?
* Estimated time to receive a new card.
* Temporary alternatives - is there emergency approval while waiting?
* Any changes to the process depending on the issuing country.
* Relevant authorities or helplines for further assistance.
  + - 1. **בנו זחלן המחזיר את התוצאות לשאילתות שהגדרתם. מומלץ (אך לא חובה) לבנות את הזחלן בפייתון. הציגו את קוד הזחלן.**

**תהליך ה-Crawling ביצוע crawling עבור עד 100 דפים ובחירת ה 50 הדפים הרלוונטיים ביותר עבור השאילתה**

בתהליך ה-Crawling מימשנו מנגנון לסריקה ממוקדת של דפי אינטרנט, במטרה לאתר 50 דפים עבור השאילתה. תהליך זה כלל סינון, אינדוקס ודירוג של הדפים שנסרקו, כדי להבטיח תוצאות איכותיות.

**פירוט התהליך:**

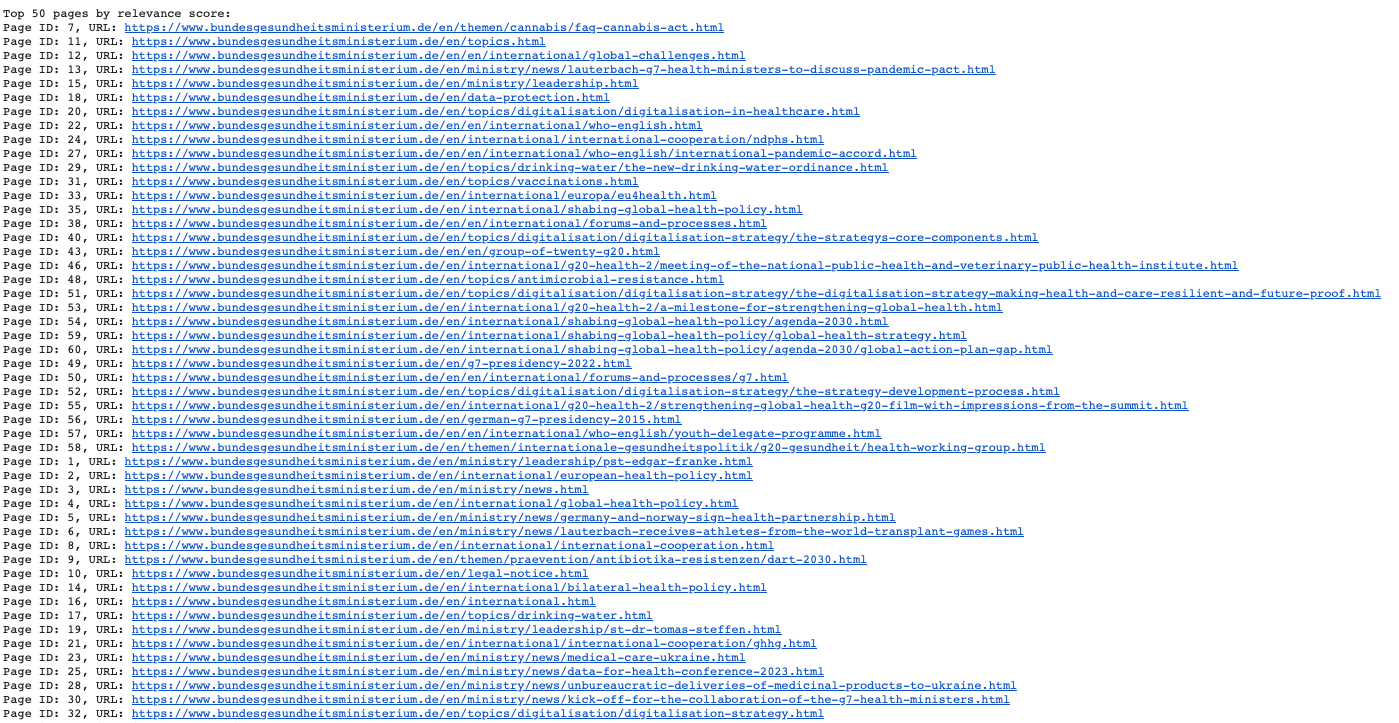
1. **התחלת תהליך הסריקה**: התהליך מתחיל עם URL ראשוני שנמסר כנקודת התחלה לסריקה. יצרנו רשימה "לביקור" הכוללת עמודים שעדיין לא נסרקו ורשימת "מבוקרים" שבה נשמרו עמודים שכבר סרקנו.
2. מהעמוד הראשוני, כל הקישורים הקיימים נשלפים (באמצעות הפונקציה get\_links) וקישורים אלו יוצרים את רשימת הכתובות שעדיין צריך לסרוק (to\_visit).
3. **סינון מקדים:** מימשנו מנגנון סינון מקדים של קישורים. מנגנון זה יזהה ויסנן דפים שאינם רלוונטיים כגון עמודי מידע מנהלתיים, עמודי יצירת קשר או תנאי שימוש, ובכך יצמצם את היקף העיבוד לתוכן המהותי בלבד.
4. **סריקת דפים**:
   * נשלחה בקשת HTTP לכל URL ברשימת "לביקור", ותוכן ה-HTML של העמוד הושג ושמור.
   * תוכן הדף עבר עיבוד באמצעות ספריית BeautifulSoup כדי לחלץ טקסט וקישורים רלוונטיים.
   * עבור כל דף תודפס הודעה עם כתובת העמוד ומזהה ייחודי (doc\_id).
5. **ניתוח תוכן הדף**:
   * חילוץ מילים מתוך טקסט העמוד וביצוע ניקוי ראשוני, כולל הסרת מילים נפוצות (Stop Words) וביצוע Stemming למילים באמצעות PorterStemmer.
   * חילוץ תגיות <title> ואת הכותרות (h1, h2, h3) מה-HTML של הדף.
   * תוכן העמוד מתווסף לאינדקס באמצעות הפונקציה index\_words, והתוצאה היא מילון page\_index המתאר את מספר הפעמים שכל מילה מופיעה בעמוד.
6. **בחינת רלוונטיות**:
   * אם הדף מכיל מונחי שאילתה מחושב עבורו ציון רלוונטיות.
   * לכל עמוד חישוב ציון הרלוונטיות נעשה באמצעות חישוב ציון TF-IDF מנורמל המבוסס על תדירות הופעת המילים (TF) ומשקל הפוך להופעתן במסמכים אחרים (IDF). מונחים שהופיעו בכותרות או בכותרות משנה קיבלו משקל מוגבר.
   * לבסוף, המידע עבור הדף (מזהה הדף וציון הרלוונטיות של העמוד) מתווסף לרשימת הדפים הפוטנציאליים.
7. **חילוץ קישורים נוספים**:
   * קישורים נוספים מתוך הדף נוספו לרשימת "לביקור", תוך סינון עמודים שכבר נסרקו או אינם רלוונטיים לתחום החיפוש.
8. **מיון ושמירת התוצאות**:
   * ציון הרלוונטיות של כל עמוד שנסרק שימש למיון הדפים. מתוך כלל הדפים שנסרקו, הוחזרו 50 הדפים שממוינים בסדר יורד של ציון הרלוונטיות.
9. **תוצאה סופית**:
   * 50 הדפים שנבחרו מציגים את השאילתה בצורה הטובה ביותר, בהתבסס על מילות החיפוש, הקשרן, והמידע הרלוונטי שנמצא בכל עמוד.

**הפלט שקיבלנו:**

**סריקת הדפים:**

****

**בחירת ה50 דפים הרלוונטיים ביותר:**

****

**הקוד: מצורף בסוף הקובץ.**

* + - 1. **רשמו טכנולוגיות שונות מעניינות שהשתמשתם בהם בפרויקט.**

**טכנולוגיות מעניינות שבהן השתמשנו בהם בפרויקט:**

**1. BeautifulSoup-** טכנולוגיה לעיבוד דפי HTML לצורך חיפוש קישורים וניתוח תכנים מתוך דפי אינטרנט, מאפשרת למצוא תגיות וקישורים בדפי אינטרנט.

**2. NLP עם nltk -** משתמשים ב-nltk כדי לבצע עיבוד שפה טבעית, ובפרט שימוש ב-PorterStemmer לצורך חיתוך מילים (stemming) שמקטין מילים לשורש שלהן, דבר שיכול לסייע בשיפור ביצועים בתהליך חיפוש.

**3. Requests-** שימוש בספריית requests לשליחה וקבלת בקשות HTTP, כדי לאסוף נתונים מדפי אינטרנט, כולל הבאת תוכן של דפי אינטרנט.

**4. ספריית Pandas -** ליצירת DataFrames ולניהול נתונים בצורה מבנית, כך שניתן לשמור את התוצאות לקובץ Excel ולבצע ניתוחים וייצוא נתונים בצורה נוחה.

**5. Time Delay -** הקוד של הזחלן משתמש ב- time.sleep() כדי להימנע מהעמסת יתר על אתרי אינטרנט במהלך הזחילה והשאילתות בכדי שלא נחסם.

**6. ספריית re- Regular Expression :** זהו כלי עוצמתי לחיפוש ועיבוד טקסט מבוסס דפוסים, בעזרת re.findall(r'\w+', text) ביצענו חילוץ מדויק של מילים וניתוח תוכן עמודי האינטרנט בצורה גמישה.

**7. ספריית defaultdict:** מאפשר עבודה נוחה עם מילונים בעזרת defaultdict(int) ביצענו אחסון וספירה של המילים השכיחות ביותר בכל מסמך.

**8. שימוש בספריית Math** לצרוך ביצוע חישובים שונים כמו חישובי הIDF לכל מילה כחלק מחישוב המדד TF-IDF . ספרייה זאת מאפשרת חישוב מדויק של משקל מילים במסמכים, על פי תדירות הופעתן**.**

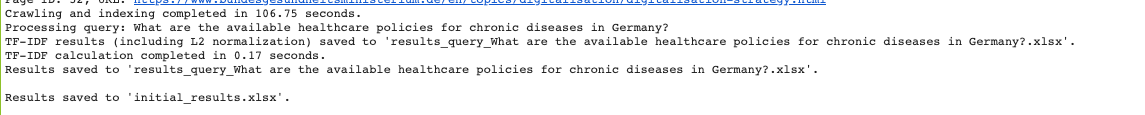
**9. Excel Writer עם Openpyx:** מנוע כתיבה ועריכה של גיליונות Excel כחלק מספריית Pandas. ייצאנו את תוצאות הניתוח, מיפוי ה URL, ספירת המילים, חישובי הTF-IDF לפורמט Excel, מה שסיפק לנו תוצר ברור ומסודר להצגת התוצאות.

**10. IPython Display and Widgets** - השתמשנו בווידגטים ליצירת ממשקי משוב אינטראקטיביים והצגת התוצאות במחברת Jupyter כחלק מהתהליך בקשת משוב רלוונטי/לא רלוונטי מהמשתמש לדפים שחזרו.

**11. NumPy -** משמש לפעולות וקטוריות במערכת משוב הרלוונטיות וחישובי דמיון קוסינוס ועבור ווקטורים הTF-IDF.

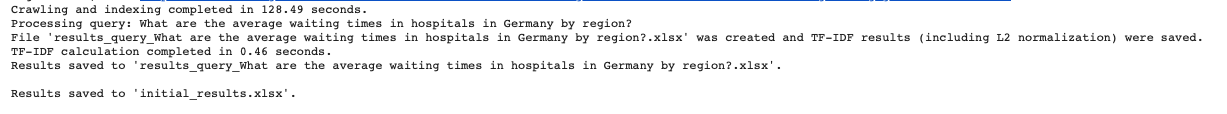
* + - 1. **כמה זמן רצו השאילתות שלכם?במה זה תלוי? האם לדעתכם ניתן לשפר זמן זה?**

**עבור השאילתה 1:**

****

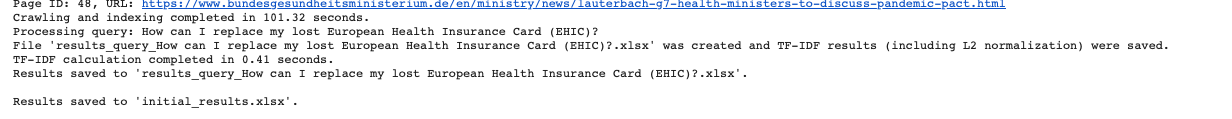
סה״כ זמן ריצה : 106.75+0.17=106.92 שניות.

**עבור שאילתה 2:**



סה״כ זמן ריצה : 128.49+0.46=128.95 שניות.

**עבור שאילתה 3:**

****

סה״כ זמן ריצה : 101.32+0.41=101.73 שניות.

**גורמים המשפיעים על זמני הריצה**

**מבנה האתר:**

* אתרים עם מבנה קישורים מורכב הכולל קישורים פנימיים רבים או דפים מקשרים זה לזה באופנים עמוקים יגרמו לזחלן לבצע יותר עבודה, מאחר שהוא צריך לאתר ולעבד מספר רב של קישורים.
* העומק שאליו הזחלן נדרש להגיע גם משפיע. אם הוא צריך לסרוק דפים רבים במבנה עץ עמוק, הזחילה תיקח זמן רב יותר.
* באתרים עשירים בקישורים קיים תוכן רב וכתוצאה מכך זמן עיבוד הטקסט והוספתו לאינדקס עולה.

**מספר הדפים שנסרקים:**

* מספר הדפים משפיע ישירות על זמן הריצה. בפרויקט הוגדרו 50 דפים כגבול, ולכן זמן הסריקה מוגבל למספר הזה. אם נעלה את המספר, זמני הריצה יגדלו פרופורציונלית.

**עיבוד טקסט TF-IDF ו- Stemming:**

* עיבוד טקסט מהווה חלק חשוב, אך הזמן הנדרש תלוי בגודל האינדקס ובכמות המילים בטקסטים.
* סינון מילים לא רלוונטיות - Stop Words ויישום Stemming דורשים עיבוד חישובי, אך הוא מוגבל בהיקפו כי הוא פועל רק על הטקסט שנסרק.
* למרות שהחישוב דורש זמן, זהו אחד השלבים המהירים יחסית, כיוון שהוא מתבצע על טקסט שכבר נמצא בזיכרון ולא דורש אינטראקציות נוספות עם השרת.

**תשובת השרת ואיכות חיבור האינטרנט:**

* זמן התגובה של השרתים באתר שסורקים משפיע ישירות על זמני הזחילה.
* אם השרת איטי או אם יש בעיות בחיבור האינטרנט, ייתכן שיהיה צורך להמתין יותר זמן כדי לקבל תשובה מהשרת.

**PageRank:**

* חישוב PageRank תלוי במספר הקישורים ובמבנה הקישורים. ככל שיש יותר קישורים, החישוב נהיה מורכב יותר.
* מספר האיטרציות שנדרש כדי להגיע להתכנסות משפיע באופן ישיר על זמן החישוב.

**Feedback Relevance:**

* תהליך דירוג מחדש באמצעות Rocchio תלוי במספר המסמכים שסומנו כרלוונטיים או לא רלוונטיים.
* אורך הווקטורים של TF-IDF משפיע על הזמן הנדרש לחישוב Cosine Similarity, ככל שיותר מילים נכללות, כך זמן החישוב גדל.
* מספר האיטרציות של הדירוג מחדש יכול להוסיף לזמני הריצה אם נעשות התאמות חוזרות ונשנות.

### הצעות לשיפור זמני הריצה:

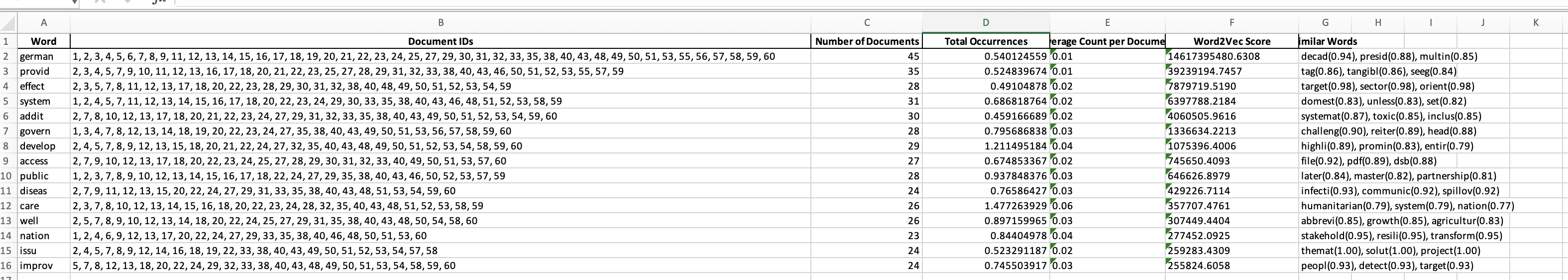
1. ניתן לממש מנגנון זחילה מקבילי ומבוזר שיחליף את המערכת הסדרתית הקיימת. מעבר לארכיטקטורה מקבילית יאפשר עיבוד בו-זמני של מספר דפים ויקצר משמעותית את משך הסריקה הכולל.
2. במהלך זחילה ישנם מקרים שבהם אותו עמוד נסרק מספר פעמים תחת כתובות שונות. בכדי להפחית עומסים מיותרים ניתן לזהות דפים זהים המופיעים תחת כתובות שונות ולמנוע עיבודים חוזרים מה שיתרום לחיסכון בזמן ובמשאבי מערכת.
3. לפעמים שאילתות קצרות אינן מכסות את כל ההקשרים האפשריים, על ידי הוספת מילים נרדפות או ביטויים קשורים לשאילתה, ניתן לשפר את תוצאות החיפוש ולהקטין את הצורך בחיפושים חוזרים.
   1. **בנו inverted index ל- 15 המילים הנפוצות שחזרו (מספיק להתיחס ל -20 הדפים הראשונים שחזרו)**

**הסבר לאופן חישוב ה-Inverted Index  עם 15 המילים הנפוצות ביותר מתוך 50 הדפים שהוחזרו:**

1. **איסוף המילים מהדפים שנבחרו**:
   * סרקנו ובחרנו את הדפים הרלוונטיים ביותר (עד 50 דפים) בהתבסס על רלוונטיות לשאילתה ע״י שימוש בTF-IDF, כאשר הדפים שנסרקו מיוצגים כמילון המכיל את תדירויות המילים בכל דף.
   * ביצענו סינון למילים שהם stop words.
2. **יצירת מודל Word2Vec:**
   * אימנו את המודל Word2Vec על בסיס הדפים שנסרקו, תוך שמירה על קשר סמנטי בין המילים.
   * לכל דף יצרנו רשימה של מילים חוזרות בהתאם לתדירות שלהן, והשתמשנו במודל לאימון על הנתונים.
3. **חישוב TF-IDF מילים כלליות**:
   * לכל מילה בדף אנחנו מחשבים את הווקטור TF\_IDF= TF\*IDF.
   * לאחר חישוב הערך TF\_IDF אננו מנרמלים אותו בעזרת L2 Normalization כאשר כל ערך בווקטור מחולק בגודל הכולל של הווקטור, כדי להבטיח נרמול ובכך לאפשר השוואה הוגנת בין מסמכים באורכים שונים.
4. **חישוב חשיבות המילים וסינון 15 המילים המשמעותיות ביותר:**
   * עבור כל מילה, ציון החשיבות שלה מחושב על סמך:
     + TF-IDF מנורמל במסמכים שבהם היא מופיעה.
     + דמיון סמנטי למילים אחרות באמצעות Word2Vec לאחר חישוב הערך.
     + חשיבות המילה = TF-IDF המנורמל + דמיון סמנטי.
   * מתוך כל המילים בחרנו את 15 המילים עם החשיבות הגדולה ביותר.
5. **בניית ה־Inverted Index החדש**:
   * בנינו Inverted Index  חדש המכיל רק את 15 המילים הנפוצות ביותר שנבחרו כאשר עבור כל מילה נשמרים הדפים שבהם היא מופיעה והערך שלה לאחר נרמול.

**הפתרון נמצא תחת לשונית Top 15 Inverted Index בקובץ אקסל initial\_results.**

**\*הריצה היא על 50 הדפים הרלוונטיים ביותר.**

****

**קוד התוכנית : נמצא בסוף הקובץ כחלק מהקוד של הCrawler.**

* 1. **בחרו את אחת השאילתות, וחשבו tfIdf של המושגים בשאילתא.**

**השאילתה היא:**

"What are the available healthcare policies for chronic diseases in Germany?"

**שלב ראשון:**

פירוק לtoken :

[What, are, the, available, healthcare, policies, for, chronic, diseases, in, Germany]

**שלב שני:**

אחרי הורדת Stop words:

[available, healthcare, policies, chronic, diseases, Germany]

**שלב שלישי:**

Stemming:

[available, healthcare, policy, chronic, disease, Germany]

**שלב רביעי:**

ביצענו את חישובי הTF-IDF בקוד כולל פירוט השלבים כתיעוד והתוצאה נשמרה בקובץ אקסל בלשונית QUERY TF-IDF.

**חישוב הTF-IDF מתבצע באופן הבא:**

**חישוב TF (Term Frequency)** :

ה-TF מחושב כדי לייצג את תדירות המילה במסמך מסוים.

* לכל מילה w  בעמוד

A close up of a text

Description automatically generated

**חישוב DF (Document Frequency)**:

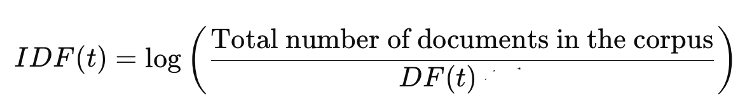
* המספר הכולל של העמודים המכילים את המילה w.



**חישוב IDF (Inverse Document Frequency)**:

* ה-IDF מחושב כדי לייצג עד כמה המילה נפוצה או ייחודית בקורפוס המסמכים כולו.





**חישוב TF-IDF**:

* ה-TF-IDF משלב את שניהם לציון המייצג את חשיבות המילה הן ברמת המסמך הבודד והן ביחס לשאר המסמכים.

A black text with a white background

Description automatically generated

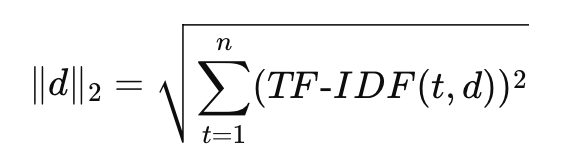
* 1. **נרמול ציוני הTF-IDF:**

נרמול (Normalization) של ציוני TF-IDF  נועד לוודא שהשוואת מסמכים תהיה הוגנת ואינה תלויה בגורמים כמו אורך המסמך או סכום הערכים שלו. זה מאפשר למדוד רלוונטיות יחסית בין מסמכים ושאילתות.

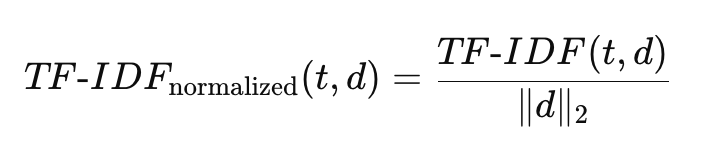
* L2 normalization:

מחלקים כל ערך TF-IDF באורך הווקטור של המסמך.

**אורך הווקטור של המסמך:**

****

**לאחר מכן אנחנו מנרמלים כל רכיב:**

****

* + - 1. **יישום בקוד**:

הקוד מחשב TF−IDF  על בסיס שאילתה נתונה:

* עובר על כל עמוד ב page\_indexes.
* מחשב TF-IDF לכל מילה בשאילתה.
* נרמול של ערכי הTF-IDF.

**את הפלט המלא ניתן לראות תחת הקובץ אקסל results\_query\_What are the available healthcare policies for chronic diseases in Germany? בלשונית Query TF-IDF:**



* 1. **האם בדפים שהוחזרו קיימים hubs? Authorities? נמקו.**

**כפי שנלמד בכיתה:**

**Hubs:** הוא דף שמכיל הרבה קישורים איכותיים לדפים אחרים בנושא.

את הHubs ניתן לזהות ע״י :

* בדיקת המידע שנאסף ע״י פונקציית get\_links() עבור כל URL.
* הדפים עם מספר הקישורים היוצאים הגבוהה ביותר הם פוטנציאליים להיות Hubs.

**Authorities:** הוא דף שמקבל הרבה קישורים מדפים אחרים איכותיים כלומר הוא דף שאליו מפנים הרבה Hubs אחרים.

בקוד ניתחנו את מבנה הקישורים של הדפים שנאספו באמצעות אלגוריתם HITS. האלגוריתם מזהה דפים שהם**Hubs**  שמפנים להרבה**Authorities** ו־**Authorities** שאליהם מפנים הרבה **Hubs** .

### שלבי התהליך:

#### ****בניית מבנה הקישורים****:

* הקוד בונה **גרף קישורים** מהדפים שנסרקו:
  + **outgoing\_links**: עבור כל דף, נשמרים הדפים שהוא מפנה אליהם.
  + **incoming\_links**: עבור כל דף, נשמרים הדפים שמפנים אליו.
* הפונקציות fetch\_page  ו־get\_links  משמשות לשליפת דפי HTML וקישורים מהם.

#### ****אתחול ציון Hubs ו־Authorities****:

* לכל דף מוקצה ציון ראשוני של 1.0  עבור**Hub**  ו־**Authority**.

#### ****חישוב ציונים באמצעות איטרציות כאשר בכל איטרציה מבוצעים השלבים הבאים****:

* **עדכון ציון Authority**:
  + הציון של דף נקבע כסכום ציוני ה-Hubs של הדפים שמפנים אליו.

auth\_scores[page] = sum(hub\_scores[incoming] for incoming in incoming\_links[page])

* **נרמול ציון Authority**:
  + כל הציונים מנורמלים ע״י חישוב הנורמה של כל הציונים, וחלוקת כל ציון בנורמה.
  + הנורמליזציה לכל ציון מתבצעת כדי למנוע גדילה לא פרופורציונלית של ערכים באיטרציות רבות.
* **עדכון ציון Hub**:
  + הציון של דף נקבע כסכום ציוני ה־Authorities של הדפים שהוא מפנה אליהם.

auth\_scores[page] = sum(hub\_scores[incoming] for incoming in incoming\_links[page])

* **נרמול ציון Hub**:
  + כל הציונים מנורמלים ע״י חישוב הנורמה של כל הציונים, וחלוקת כל ציון בנורמה.
  + הנורמליזציה לכל ציון מתבצעת כדי למנוע גדילה לא פרופורציונלית של ערכים באיטרציות רבות.

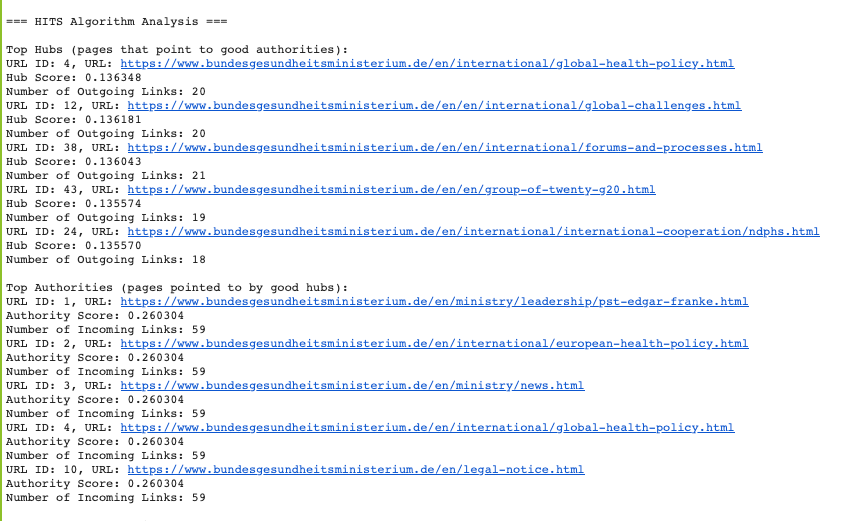
#### ****בדיקת התכנסות****:

* אם ההפרשים בין הציונים החדשים לציונים הישנים בין שתי איטרציות רצופות עבור כל הדפים קטנים מסףϵ האלגוריתם מפסיק את האיטרציות.

#### ****תוצאה סופית****:

* האלגוריתם מדפיס את חמשת הדפים המובילים כ־**Hubs**  וכ־**Authorities** על פי הציונים הסופיים עם המידע הבא:
  + URL ID – מזהה הדף
  + כתובת URL מלאה.
  + ציון ה- Hub אוAuthority כלומר מדד איכותי המשקלל את חשיבות הדף במבנה הרשת
  + מספר הקישורים היוצאים או נכנסים.
* דף נחשב לHub טוב לא רק אם יש לו הרבה קישורים יוצאים, אלא אם הוא מקשר לדפים שהם Authorities טובים.
* דף נחשב לAuthority טוב לא רק אם יש לו הרבה קישורים נכנסים, אלא אם הוא מקושר

**תוצאות הרצת האלגוריתם:**

****

* 1. **בחרו 10 דפים שהזחלן החזיר, ואשר יש ביניהם קישורים . חשבו pageRank לכל דף, מומלץ להעזר בסקריפט מוכן. הציגו את החישובים ואת הדירוג הסופי.**

השתמשנו בסקריפט המוכן מההרצאה לחישוב הPageRank לכל דף ושידרגנו אותו מעט.

**השלבים לחישוב לחישוב הPageRank לכל דף:**

* **בנייה ראשונית של מבנה הקישורים**:
  + הפונקציהselect\_connected\_pages  יוצרת גרף קישורים על בסיס הקישורים שנמצאים בתוך הדפים:
  + page\_links - עבור כל דף, נרשמים הדפים שאליהם הוא מפנה.
  + כל דף חייב לכלול קישורים יוצאים לפחות לדף אחד כדי להיכלל במבנה.
* **בחירת 50 דפים עם קשרים ביניהם**:
  + הפונקציה מתחילה לסדר את הדפים לפי מספר הקישורים היוצאים בסדר יורד.
  + הדפים הראשונים שנבחרים הם אלו שיש להם יותר קשרים.
  + במהלך הבחירה, הפונקציה מבטיחה שכל דף שנבחר הוא חלק מקבוצת דפים מחוברת connected component.
* **חישוב המבנה המחובר**:
  + הפונקציה בודקת קשרים בין דפים שנבחרו כדי ליצור גרף מקושר סופי.
  + עבור כל דף שנבחר, היא שומרת את הקישורים היוצאים שמתייחסים לדפים אחרים בתוך 50 הדפים הנבחרים.

**חישוב הPage Rank עבור כל דף:**

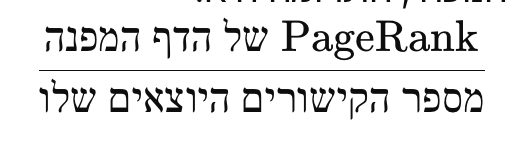
* **אתחול ציוני PageRank**:
* לכל דף מתוך ה-50 הנבחרים מוקצה ציון התחלתי שווה

כאשרN הוא מספר הדפים שנבחרו =50

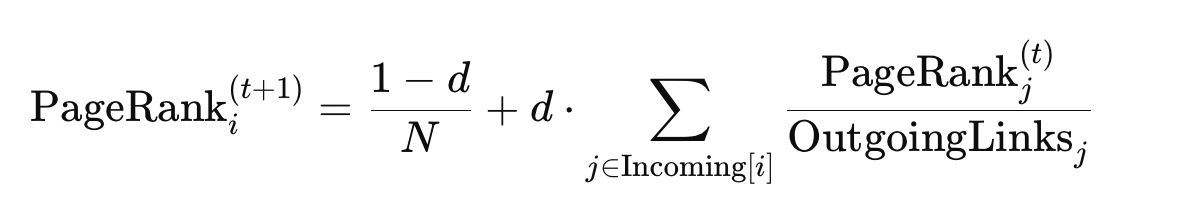
* **לולאת עדכון PageRank:**
* הציונים מתעדכנים לאורך מספר איטרציות שנקבע על ידי num\_iterations  או עד להתכנסות.
* בכל איטרציה מבוצע עדכון לכל דף באופן הבא:
  + **קלטים נכנסים**: נחשב את הסכום של התרומות מ־PageRank של הדפים שמפנים לדף הנוכחי.
  + **תרומת "קפיצה אקראית"**:

*משקל שמחולק שווה בין כל הדפים*

* **תרומת קישורים נכנסים**:
  + עבור כל דף שמפנה לדף הנוכחי, התרומה היא:



* + הנוסחה המלאה:

**

*כאשר:*

* d- מקדם שיכוך בדרך כלל 0.85, מייצג את ההסתברות שהמשתמש ימשיך לעבור בין קישורים בדף אינטרנט, לעומת ההסתברות שהוא "יתחיל מחדש" או ידלג באופן אקראי לדף אחר.
* N-מספר הדפים הכולל, במקרה שלנו 10 דפים.
* - גורם הקפיצה האקראי, המבטיח שלכל עמוד יש דירוג עמוד בסיסי גם אם אין לו קישורים נכנסים.
* - קבוצת דפים שמקשרים לעמוד i.
* - מספר הקישורים היוצאים מעמוד j.
* - ציון הPageRank של עמוד j באיטרציה t הנוכחית.

*נוסחה זו מבטיחה שדירוג הדף של דף מושפע מדירוג העמודים של הדפים המקשרים אליו, בשקלול מספר הקישורים היוצאים מאותם דפים.*

* **בדיקת התכנסות**:
  + לאחר כל איטרציה, הקוד מחשב את ההפרש המרבי בין הציונים החדשים לציונים הישנים.
  + אם ההפרש קטן מ־ϵ הלולאה נעצרת.
* **פלט הציונים**:
  + בסוף התהליך, לכל דף מחושב ומודפס ציון ה PageRank שמייצג את חשיבותו במבנה הקישורים.
  + בסוף התהליך מחושב הדף עם הPageRank הגדול ביותר.

דירוג זה מייצג את חשיבות הדף במונחי מבנה הקישורים, אם דף מקבל קישורים מהרבה דפים חשובים כלומר הרבה דפים בעלי PageRank גבוה הוא יקבל ציון גבוה. ואם דף מקבל קישור מדף שיש לו הרבה קישורים יוצאים, התרומה לציון שלו תהיה קטנה יותר כי היא מתחלקת בין כל הקישורים היוצאים.

**פלט :**

**עבור כל דף מציג לאילו דפים הוא מפנה:**

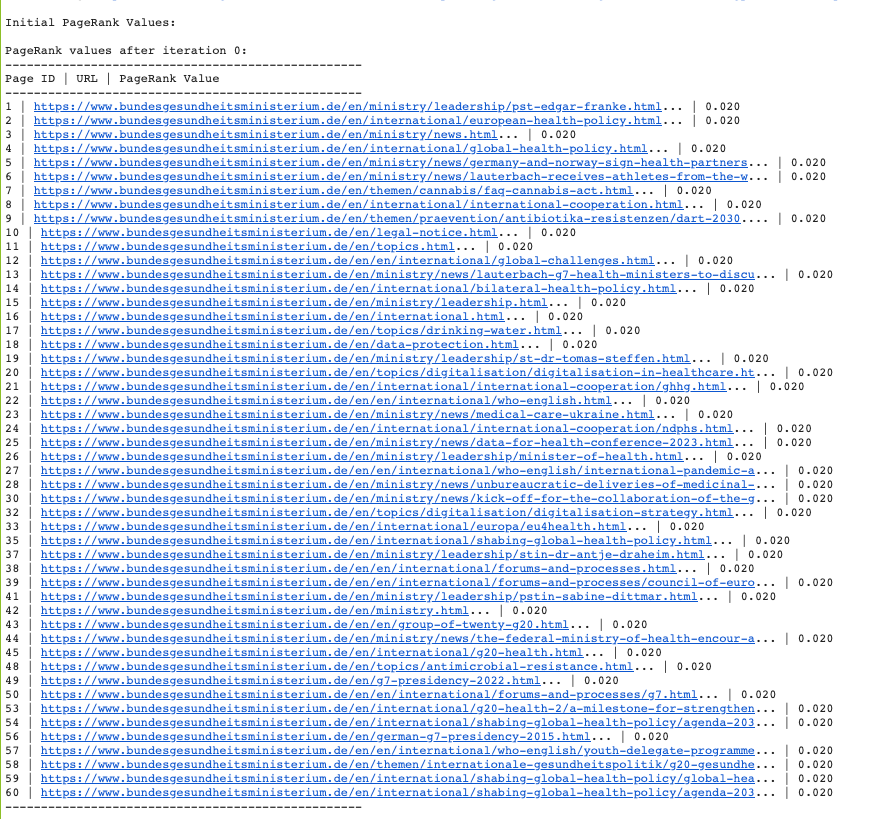




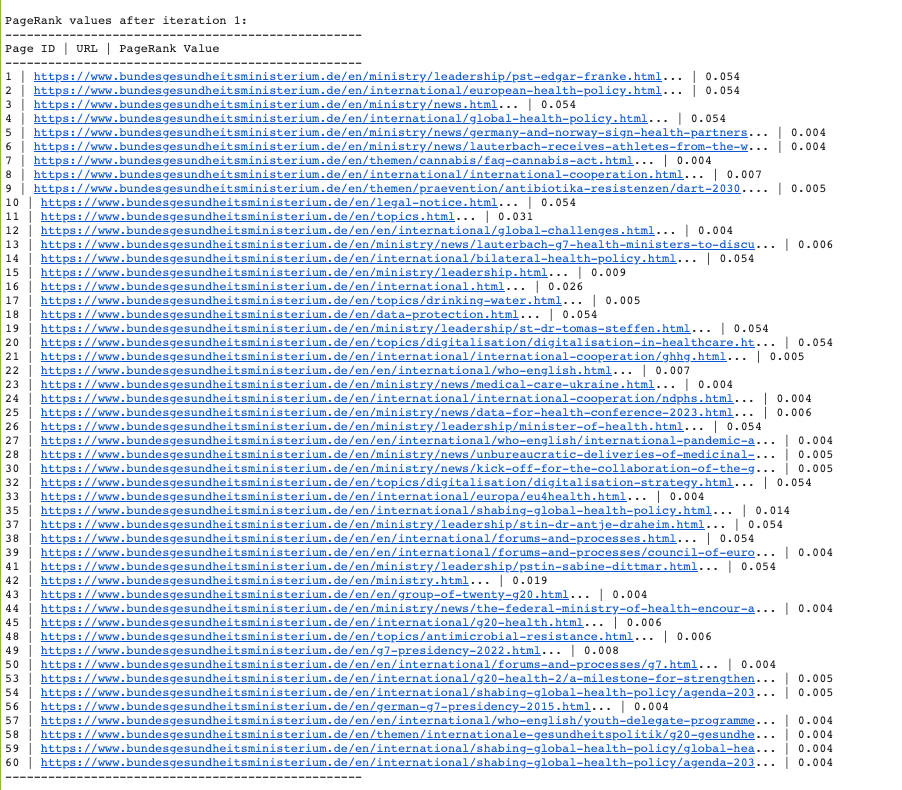


**חישוב הPageRank :**

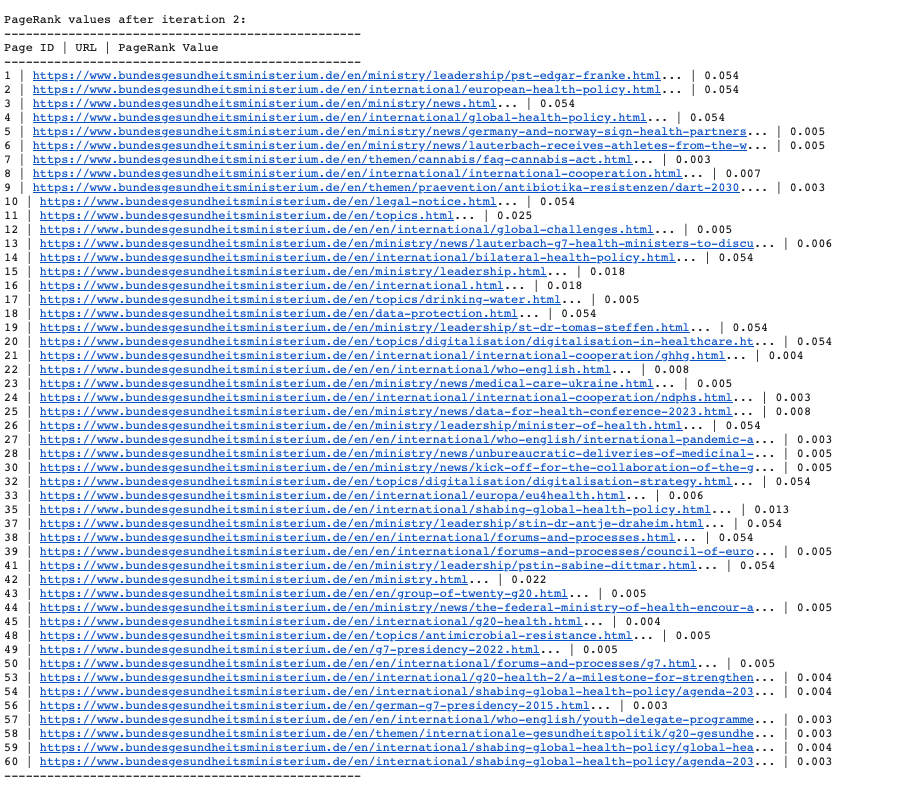
**ערך התחלתי 1/50=0.020 עבור כל דף:**



**אחרי איטרציה 1:**



**אחרי איטרציה 2:**



**הדף עם הניקוד הגבוה ביותר לאחר 2 איטרציות:**



* 1. **הראו לשני משתמשים שונים את הדירוג מהסעיף הקודם, בקשו מהם לסמן relevance feedback .**

**חשבו precision ו - recall.האם ניתן להציע שאילתא מותאמת על מנת לשפר תוצאות?**

גם פה השתמשנו בסקריפט עבור feedback relevance שהוצג בהרצאה ושילבנו אותו עם הקוד שלנו.

המימוש שלנו משתמש בעקרונות של **אלגוריתם Rocchio** לשיפור וקטור השאילתה ולדירוג מחדש של דפים על בסיס פידבק המשתמש. השימוש מתמקד בעדכון וקטור השאילתה ובחישוב דמיון קוסינוס מול דפי המערכת.

* כל משתמש מבין שני מהמשתמשים מסמן דפים כרלוונטיים או לא רלוונטיים לאחר איסוף הפידבק, המשתמש יכול ללחוץ על כפתור "Show Ranking" לדירוג מחדש של הדפים.
* **מוצגים לכל משתמש ה20 דפים עם ציון הpage rank הגבוה ביותר מתוך כל ה50 דפים שהוחזרו בזחילה.**

**תהליך הדירוג מחדש מתבצע באופן הבא:**

* + 1. **יצירת וקטורי TF-IDF**:
* נבנים וקטורי TF-IDF לכל הדפים באמצעות האינדקס ההפוך.
* מחושבים TF(תדירות המונח בדף) ו-IDF תדירות הפוכה במסמכים עבור מילים בכל דף.
  + 1. **אתחול וקטור השאילתה**:
* וקטור השאילתה הראשוני query\_vector הוא ממוצע של כל וקטורי TF-IDF של המסמכים.
  + 1. **עדכון וקטור השאילתה לפי הפידבק**:
* דפים רלוונטיים:

וקטור השאילתה מתעדכן על ידי הוספת ממוצע וקטורי TF-IDF של הדפים הרלוונטיים, משוקלל לפי

* דפים לא רלוונטיים:

וקטור השאילתה מתעדכן על ידי חיסור ממוצע וקטורי TF-IDF של הדפים הלא רלוונטיים, משוקלל לפי

* הנוסחה מתאימה לעקרונות של Rocchio:

A black and white logo

Description automatically generated

כאשר

* q\_new וקטור השאילתה המעודכן.
* q\_original וקטור השאילתה הראשוני.
* R קבוצת הדפים שסומנו כרלוונטיים.
* NR קבוצת הדפים שסומנו כלא רלוונטיים.
* פרמטרים לקביעת המשקל של רלוונטיות וחוסר רלוונטיות.
  + 1. **חישוב דמיון קוסינוס**:
* מחשבים את הדמיון הקוסינוס בין וקטור השאילתה המעודכן לוקטורי TF-IDF של כל הדפים.
* הדפים מדורגים לפי ציוני הדמיון שהתקבלו.
* הדירוגים המעודכנים מוצגים למשתמש.

**לשיפור השאילתה :**

האלגוריתם מנתח מונחים מדפים רלוונטיים ולא רלוונטיים ומציע מונחי שאילתה חדשים בהתבסס על תדירויות מונחים, כלומר המערכת לומדת מהבחירות של המשתמש כדי להציע שאילתה טובה יותר. כאשר המשתמש מסמן דף כרלוונטי המערכת מחפשת אילו מילים מופיעות הרבה בעמוד זה. מילים אלו מקבלות ״ניקוד בונוס״. לעומת זאת כשהמשתמש מסמן עמוד כלא רלוונטי המילים שמופיעות בו מקבלות ״נקודות עונש״. בסוף התהליך המערכת בוחרת את חמשת המילים עם הכי הרבה נקודות חיוביות ומשתמשת בהן כדי ליצור שאילתה חדשה. הרעיון באלגוריתם זה הוא מיקוד המערכת ע״י בחירת הרלוונטיות של המשתמש, המערכת משתמשת במידע זה כדי לנחש טוב יותר מה המשתמש מחפש.

**תהליך הרחבת שאילתה:**

המשתמש יכול ללחוץ על כפתור "Show Query Suggestion" לצורך הצעת מונחי שאילתה נוספים.

**תהליך הרחבת השאילתה**:

* + 1. **איסוף מונחים**:
* נאספים מונחים מתוך דפים רלוונטיים, והם משוקללים לפי תדירותם.
* מונחים מתוך דפים לא רלוונטיים נענשים ומופחתים מהשקלול.
  + 1. **מיון מונחים**:
* המונחים ממוקמים לפי המשקל העדכני שלהם.
* נבחרים N=5 המונחים המובילים בעלי המשקל הגבוה ביותר.
  1. **יצירת שאילתה מורחבת**:
* המונחים שנבחרו משולבים לשאילתה חדשה שמוצגת למשתמש.

**יתרונות של Feedback Relevance**:

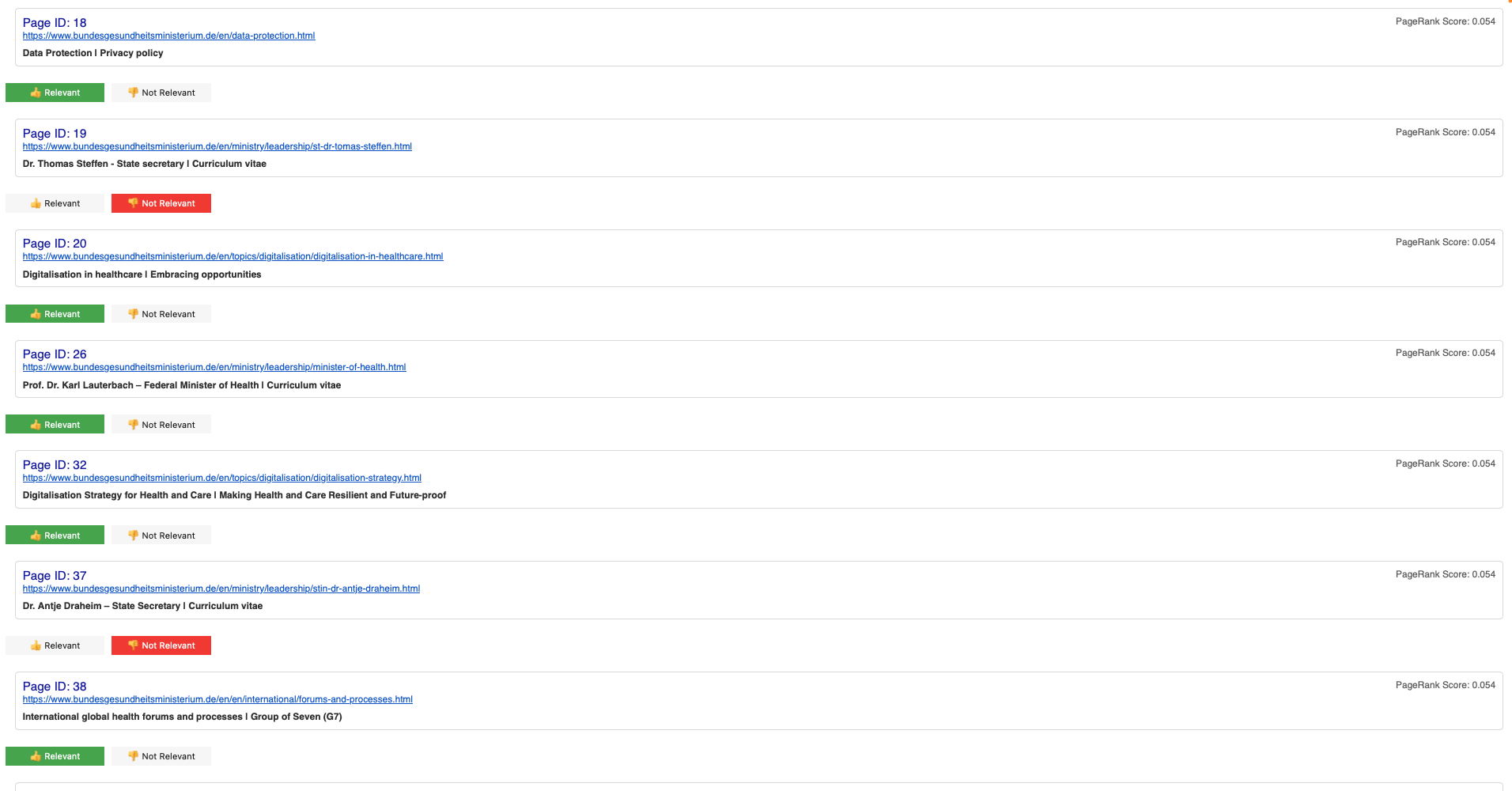
* **שיפור דירוגים**: הדירוגים מתעדכנים בהתאם לפידבק, כך שדפים רלוונטיים זוכים לעדיפות גבוהה יותר.
* **שיפור שאילתות**: הצעת מונחים נוספים יכולה לשפר את איכות התוצאות.
* **התאמה אישית**: מאפשר התאמה דינאמית של תוצאות החיפוש לצרכי המשתמש.

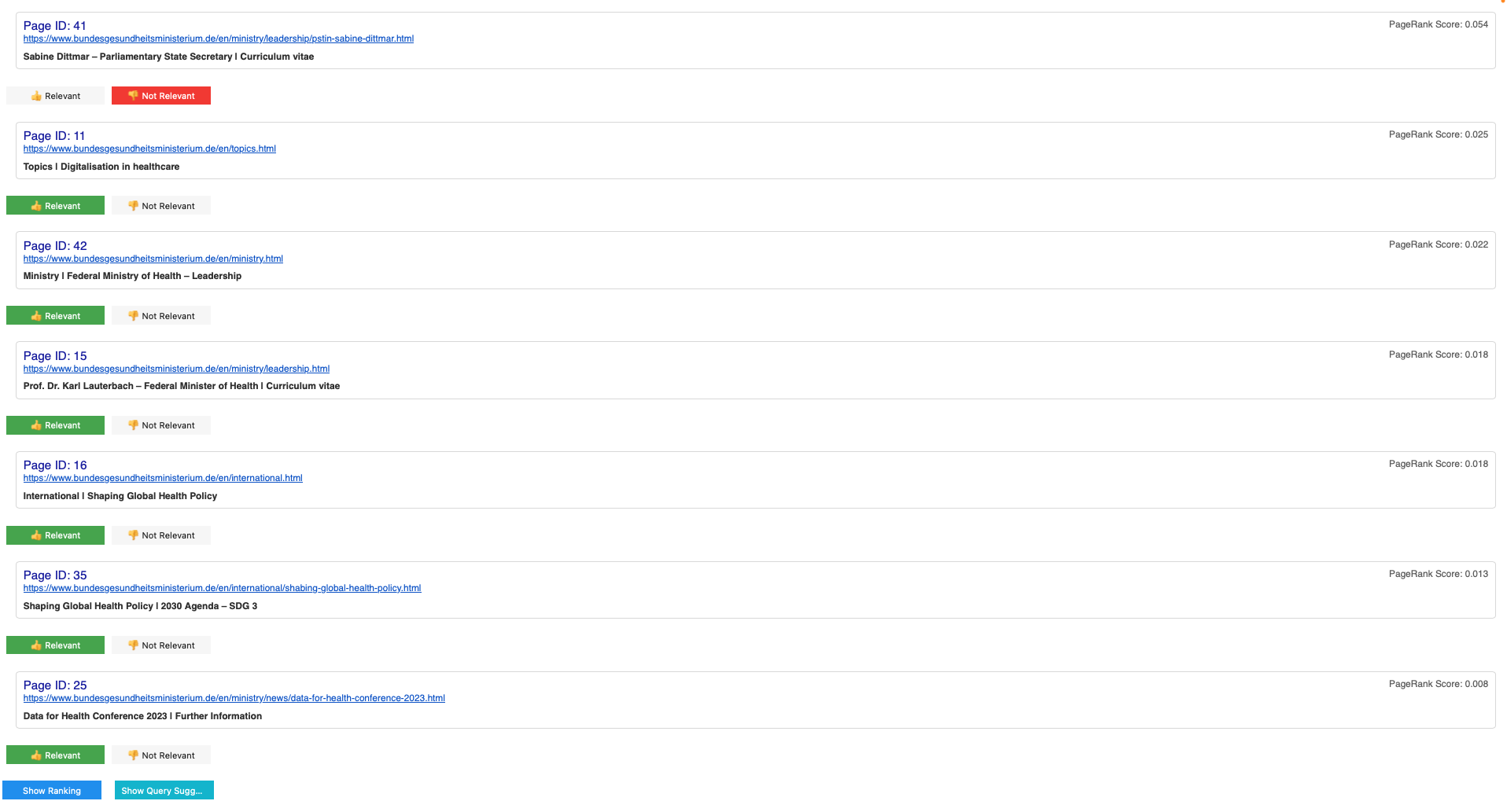
שימוש במונחי השאילתה המוצעים יאפשרו לצמצם חיפושים עתידיים ולשפר את התוצאות על סמך משוב המשתמשים.

**עבור משתמש 1 :**

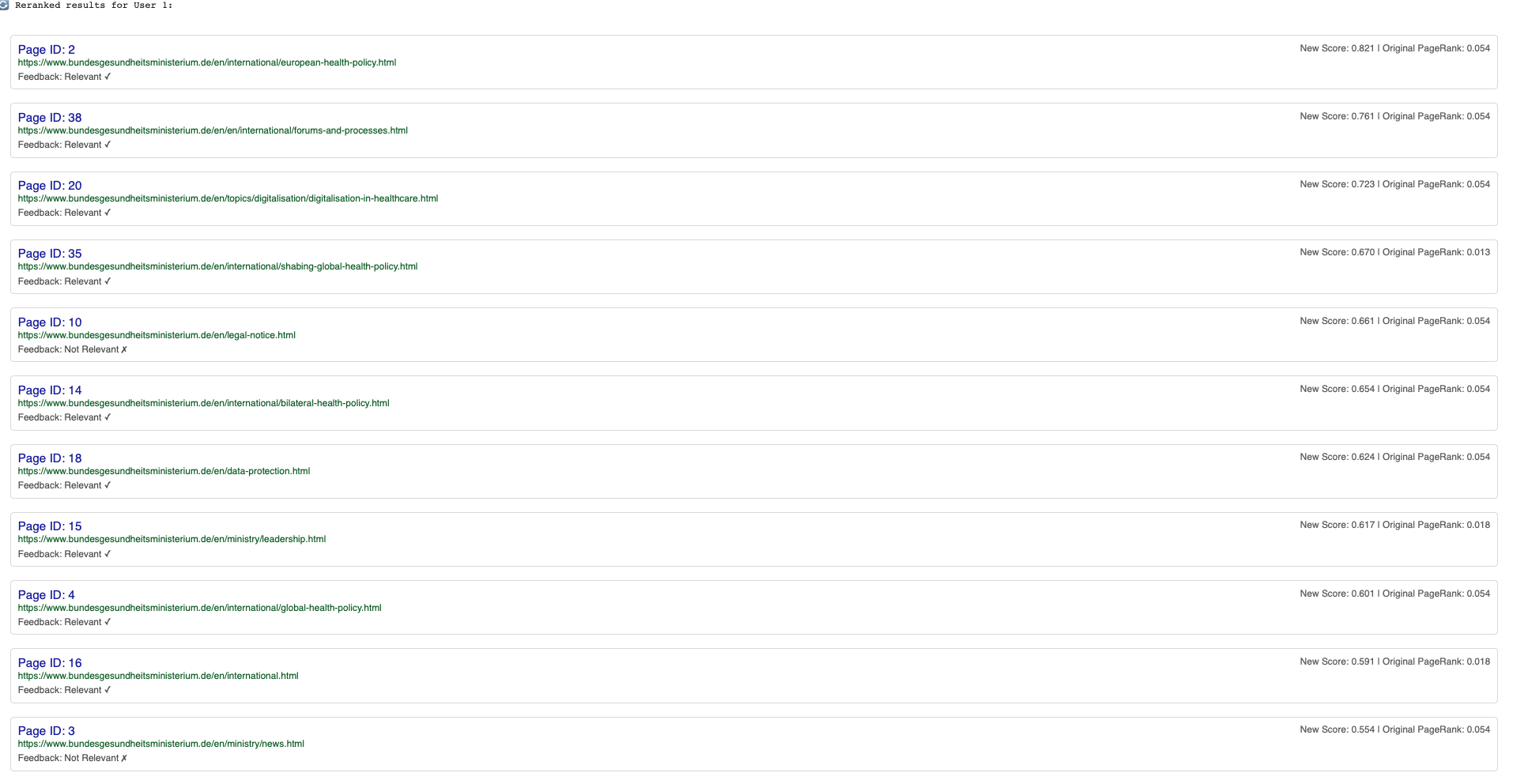
בחירות רלוונטיות/לא רלוונטיות של משתמש 1 :







**עדכון הניקוד וסדר הדפים:**





**חישוב הPrecision:**

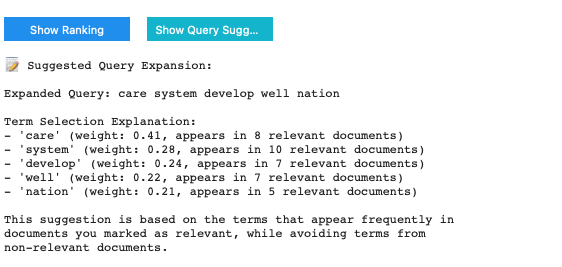
Precision=מספר התוצאות הרלוונטיות שהוחזרו/מספר כל התוצאות שהוחזרו 🡨

**חישוב ה Recall:**

Recall=מספר התוצאות הרלוונטיות שהוחזרו/מספר כל התוצאות הרלוונטיות במערכת🡨

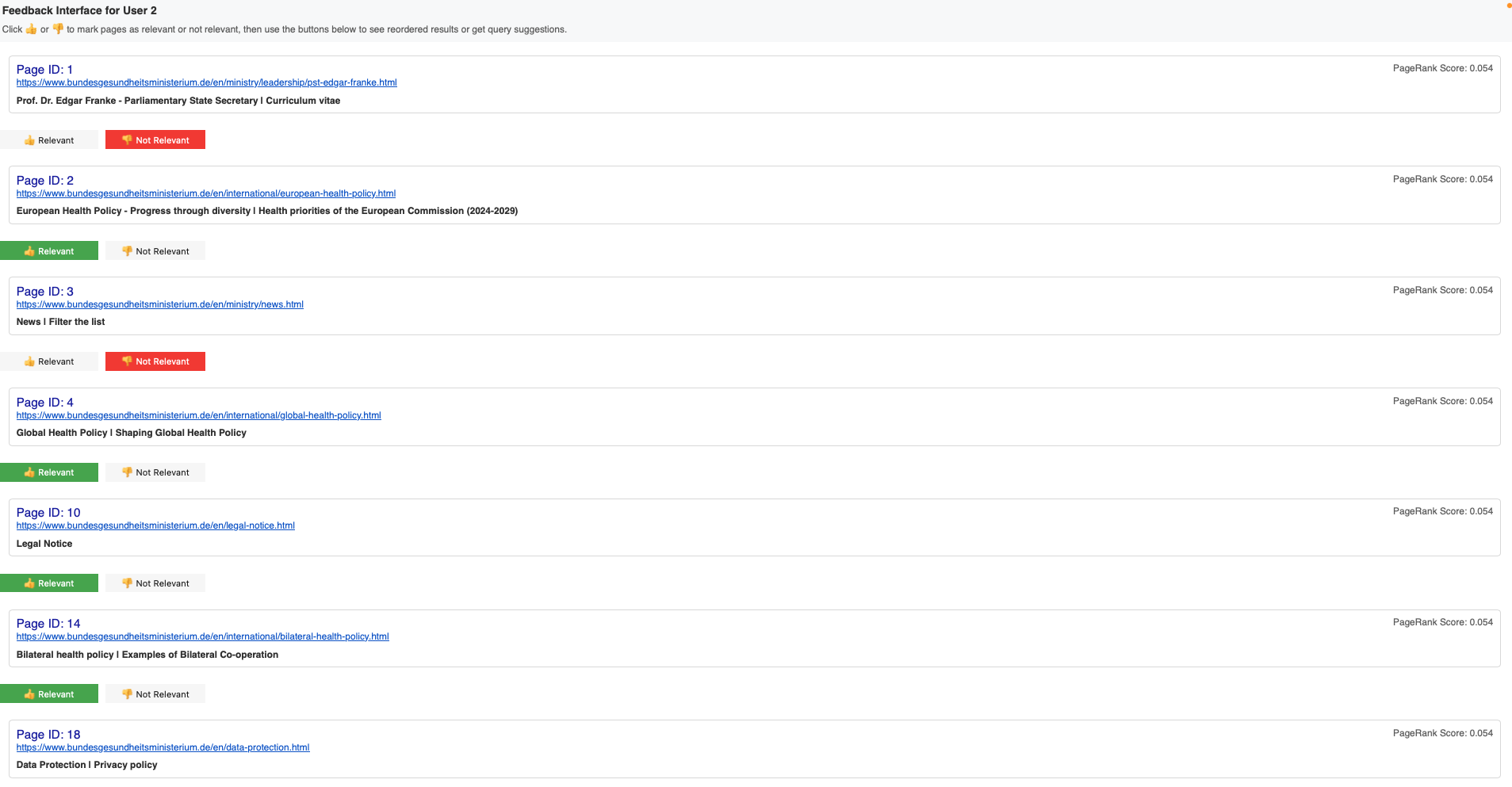
במערכת קיימים 18 מסמכים רלוונטיים לשאילתה (האתר יחסית קטן ולכן יכולנו לבדוק זאת) והמערכת החזירה 20 דפים ו-14 מתוך ה20 המשתמש סימן כרלוונטיים, מכאן הRecall:

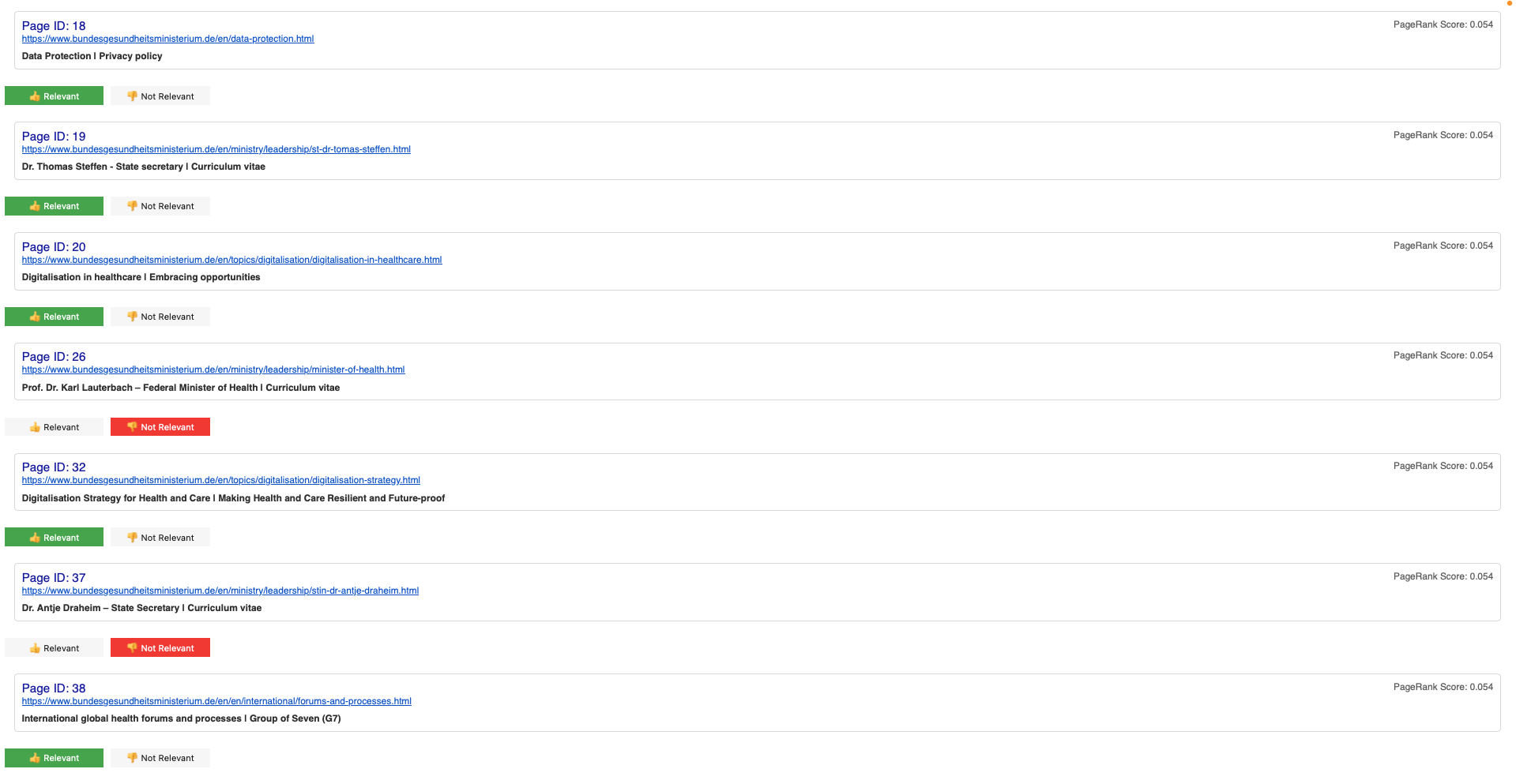
**שאילתה מותאמת על מנת לשפר תוצאות על סמך הבחירות של משתמש 1:**



**עבור משתמש 2 :**

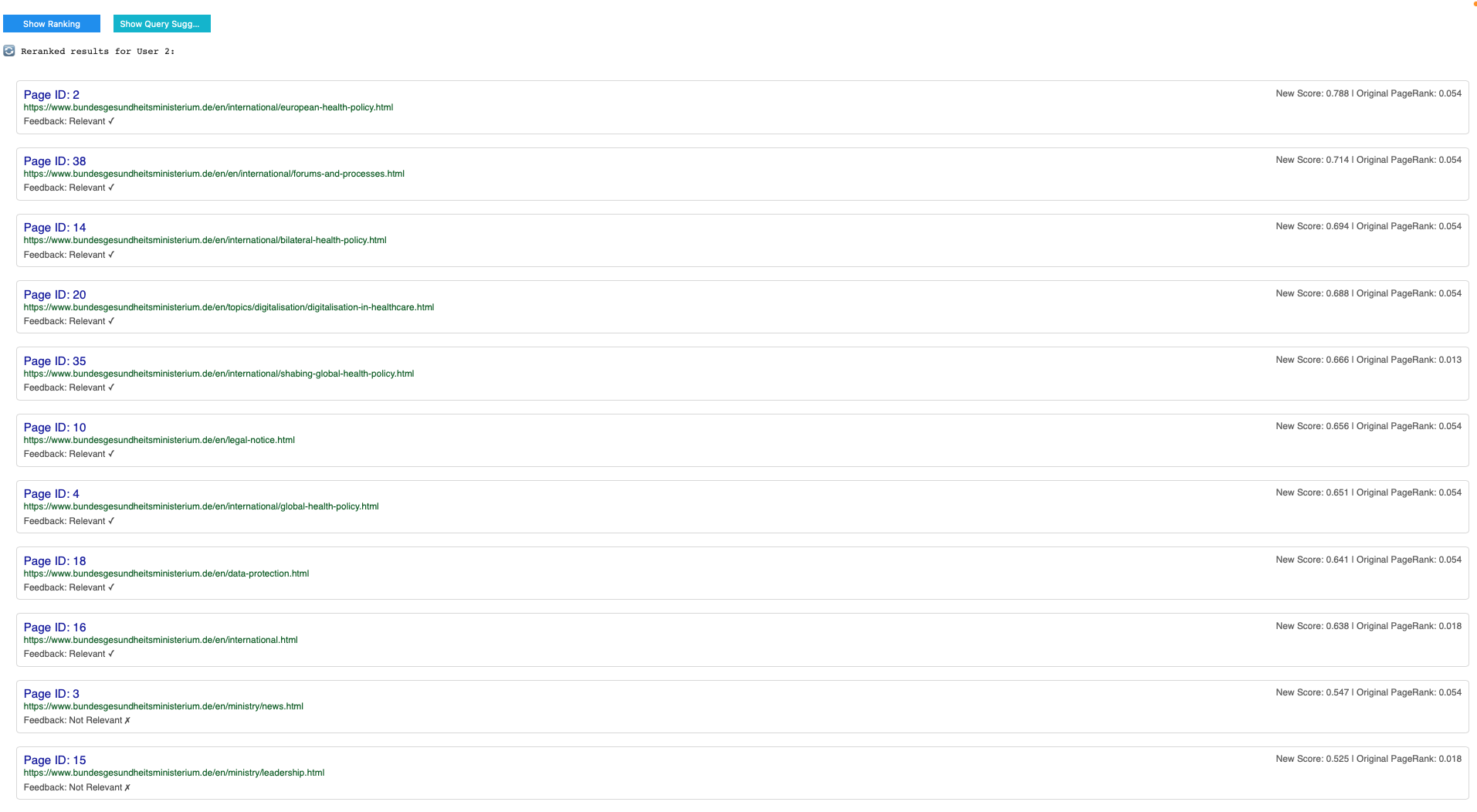
בחירות רלוונטיות/לא רלוונטיות של משתמש 2 :

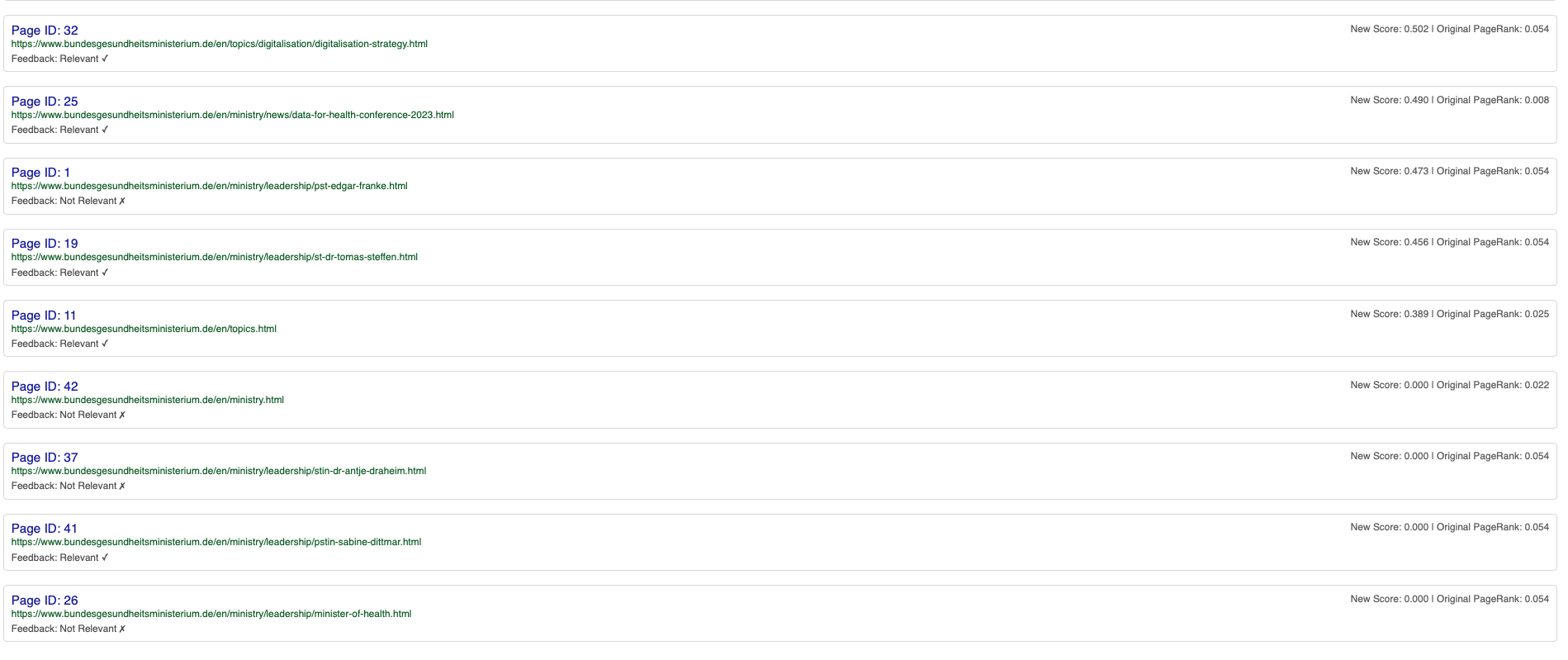






**עדכון הניקוד וסדר הדפים:**





**חישוב הPrecision:**

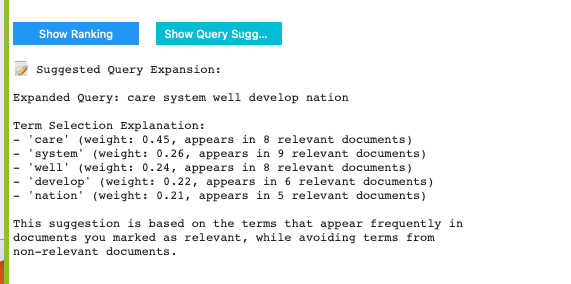
Precision=מספר התוצאות הרלוונטיות שהוחזרו/מספר כל התוצאות שהוחזרו 🡨

**חישוב ה Recall:**

Recall=מספר התוצאות הרלוונטיות שהוחזרו/מספר כל התוצאות הרלוונטיות במערכת🡨

במערכת קיימים 18 מסמכים רלוונטיים לשאילתה והמערכת החזירה 20 דפים מתוכם ו-14 מתוך ה20 המשתמש סימן כרלוונטיים, מכאן הRecall:

**שאילתה מותאמת על מנת לשפר תוצאות על סמך הבחירות של משתמש 2:**



* + - 1. **הציגו את כל התוצרים של סעיפים א-ו בדף HTML יחיד. הדף יכול להיות דינמי או סטטי (כלומר מציג צילומי מסך של התשובות לשאלות). העלו את הדף לgit pages. צרפו קישור לדף שהעליתם.**

|  |  |
| --- | --- |
| איזה שינוי הוצע? | האם לדעתכם יש מקום לשיפור זה? אם כן - הסבירו כיצד ניתן לשפר. אם לא, נמקו |
| להוסיף יותר אלגוריתמים לרלוונטיות | בהחלט אפשר להוסיף עוד אלגוריתמים כמו HITS. |
| אפשר לעשות בצורה מבוזרת כמו שאמרתן ולהוריד תוכן שהוא לא חשוב | אפשר לבנות מנגנון סינון תוכן חכם יותר, לדוגמה, להתמקד רק בדפים עם כותרות רלוונטיות לשאילתה. |
| אינדקס | השתמשנו בinverter index |
| hmmm, maybe the waiting time using multithreading | שימוש בריבוי תהליכים (multithreading) יכול להקטין את זמני ההמתנה בזחילה ובהורדת התוכן. |
| חישוב רלוונטיות לא לפני TF-IDF | אפשר לנסות להוסיף שימוש ב-Word2Vec להערכת רלוונטיות, במקום TF-IDF בלבד. |
| היית שמח לראות אם אפשר נגדי בשאילתה על מחלות כרוניות לסווג אולי לסוגי מחלות כמו נשימה/ עור וכו', | אפשרי,כדאי לשלב סיווג נושאים נוסף (למשל ניתוח לפי קטגוריות רפואיות) בהתבסס על מילות מפתח במסמכים. |
| האם אפשר לשפר את ה 30 שניות. | תמיד אפשר לשפר את זמני הריצה, למשל אם היינו משתמשים בgoogle colab pro בשביל שימוש ביותר כוח חישוב |
| בניית אינדקס רלוונטי | השתמשנו בinverter index |
| שימוש במודלים כמו BERT להבנה עמוקה של השפה והקשר בין שאילתות למסמכים. | יכול להיות פתרון טוב מאוד, במידה וקיים אפשרות לכוח חישוב חזק יותר. המודלים האלו באמת יכולים לשפר אבל מצריכים הרבה מאוד זמן עיבוד כמו שלמדנו בקורסים קודמים |
|  |  |

* + - 1. **מהו היו האתגרים בקורס/ פרויקט כצוות?**

אנחנו חושבים שהאתגר הכי משמעותי היה כתיבת הקוד, בהרצאות היו דוגמאות לקוד שאכן עזרו לנו מאוד אבל אינטגרציה של כל זה יחד לקח לנו זמן.

תרגילי הבית עזרו מאוד להבנה שלנו אבל היינו צריכים להבין יותר לעומק את הדוגמאות מהשיעור וכיצד אפשר להשתמש בהם בתרגילי הבית כמו למשל נוסחאות לחישוב ואיך להציג את התוצאות.

**קישור לGitPages :**

[**https://cohensh96.github.io/Rank-Rangers--Information-Retrieval/FinalProjectDisplay.html**](https://cohensh96.github.io/Rank-Rangers--Information-Retrieval/FinalProjectDisplay.html)

**קישור לGitHub:**

[**https://github.com/cohensh96/Rank-Rangers--Information-Retrieval**](https://github.com/cohensh96/Rank-Rangers--Information-Retrieval)

**קוד הזחלן כולל כל החישובים:**

# -\*- coding: utf-8 -\*-

"""Untitled73.ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1ci00gmjMZKlLiXsS106pHQiM\_Yt6kzHt

"""

# -\*- coding: utf-8 -\*-

"""CodeFinal.ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1b8FnEoegk3itYEwv8\_yuBpEF3uCAv4Ta

"""

# -\*- coding: utf-8 -\*-

"""Untitled23.ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1gyqSXuq-Wga83Gkhpaox72ljcg00I8WQ

"""

import requests

from bs4 import BeautifulSoup

import re

from nltk.stem import PorterStemmer

from urllib.parse import urljoin

import time

from collections import defaultdict

import math # For log in IDF

import pandas as pd

from IPython.display import HTML, display, clear\_output

import ipywidgets as widgets

import numpy as np

from sklearn.metrics.pairwise import cosine\_similarity

from gensim.models import Word2Vec

import numpy as np

from urllib.parse import urlparse, urljoin

class SearchEngine:

def \_\_init\_\_(self):

"""

Initializes the SearchEngine with a PorterStemmer, a set of stop words,

and internal mappings/data structures for indexing.

Attributes:

stemmer (PorterStemmer): Stemmer used for normalizing words.

stop\_words (set): Set of English stopwords to exclude from indexing.

url\_to\_id (dict): Maps a URL string to a unique integer ID.

id\_to\_url (dict): Maps a unique integer ID back to its URL string.

next\_id (int): Next available integer ID for a new URL.

inverted\_index (defaultdict): Maps a stemmed word to a dict of {doc\_id: count}.

global\_index (defaultdict): Similar to inverted\_index but stores {word: {url: count}}.

"""

self.stemmer = PorterStemmer()

self.stop\_words = {

'a', 'about', 'above', 'after', 'again', 'against', 'all', 'am', 'an', 'and', 'any',

'are', 'aren', 'as', 'at', 'be', 'because', 'been', 'before', 'being', 'below', 'between',

'both', 'but', 'by', 'can', 'could', 'did', 'do', 'does', 'doing', 'don', 'down', 'during',

'each', 'few', 'for', 'from', 'further', 'had', 'has', 'have', 'having', 'he', 'her',

'here', 'hers', 'herself', 'him', 'himself', 'his', 'how', 'i', 'if', 'in', 'into', 'is',

'it', 'its', 'itself', 'just', 'me', 'more', 'most', 'my', 'myself', 'no', 'nor', 'not',

'now', 'of', 'off', 'on', 'once', 'only', 'or', 'other', 'our', 'ours', 'ourselves', 'out',

'over', 'own', 's', 'same', 'she', 'should', 'so', 'some', 'such', 'than', 'that', 'the',

'their', 'theirs', 'them', 'themselves', 'then', 'there', 'these', 'they', 'this', 'those',

'through', 'to', 'too', 'under', 'until', 'up', 'very', 'was', 'we', 'were', 'what', 'when',

'where', 'which', 'while', 'who', 'whom', 'why', 'will', 'with', 'you', 'your', 'yours',

'yourself', 'yourselves','et','wh','la','à','last','also','include','chang','includ','thi'

}

# Add domain-specific terms

self.healthcare\_terms = {

'health', 'medical', 'doctor', 'patient', 'hospital',

'treatment', 'care', 'disease', 'medicine', 'therapy',

'policy', 'insurance', 'healthcare', 'clinic', 'chronic',

'diagnosis', 'prevention', 'wellness', 'specialist'

}

self.url\_to\_id = {}

self.id\_to\_url = {}

self.next\_id = 1

# inverted\_index[word][doc\_id] = count of word in doc

self.inverted\_index = defaultdict(lambda: defaultdict(int))

# global\_index[word][url] = count of word in url

self.global\_index = defaultdict(lambda: defaultdict(int))

def is\_valid\_word(self, word):

"""

Validates if a word is acceptable for indexing

"""

# Filter out strings that contain any digits

if any(char.isdigit() for char in word):

return False

# Minimum length check

if len(word) < 3:

return False

# Must contain only English letters

if not all(ord('a') <= ord(c.lower()) <= ord('z') for c in word if c.isalpha()):

return False

# Must be only alphabetic characters (no special chars)

if not word.isalpha():

return False

return True

def get\_links\_hits(self, soup, base\_url):

"""

Extracts all <a> tag hyperlinks from a BeautifulSoup page and converts them to absolute URLs.

Args:

soup (BeautifulSoup): The parsed HTML content.

base\_url (str): The base URL to resolve relative links.

Returns:

set: A set of absolute URLs found in the page.

"""

links = set()

if soup:

for link in soup.find\_all('a', href=True):

url = link['href']

absolute\_url = urljoin(base\_url, url)

if absolute\_url.startswith(base\_url):

links.add(absolute\_url)

return links

def analyze\_link\_structure(self, max\_iterations=20, epsilon=1e-8):

"""

Performs the HITS algorithm to compute hub and authority scores

for the URLs that the search engine has crawled.

Args:

max\_iterations (int): Maximum number of iterations for the HITS algorithm.

epsilon (float): Convergence threshold for stopping early.

Returns:

dict: A dictionary containing:

- 'hub\_scores' (dict): Hub scores for each URL ID.

- 'auth\_scores' (dict): Authority scores for each URL ID.

- 'outgoing\_links' (defaultdict): Mapping from URL ID to a set of outgoing link IDs.

- 'incoming\_links' (defaultdict): Mapping from URL ID to a set of incoming link IDs.

"""

outgoing\_links = defaultdict(set)

incoming\_links = defaultdict(set)

# # Build the adjacency matrix

# for url\_id in self.id\_to\_url:

# url = self.id\_to\_url[url\_id]

# soup = self.fetch\_page(url)

# if soup:

# links = self.get\_links\_hits(soup, url)

# for link in links:

# if link in self.url\_to\_id:

# target\_id = self.url\_to\_id[link]

# outgoing\_links[url\_id].add(target\_id)

# incoming\_links[target\_id].add(url\_id)

#Build initial link structure

for url\_id in self.id\_to\_url:

url = self.id\_to\_url[url\_id]

soup = self.fetch\_page(url)

if soup:

links = set()

for link in soup.find\_all('a', href=True):

absolute\_url = urljoin(url, link['href'])

if absolute\_url in self.url\_to\_id:

target\_id = self.url\_to\_id[absolute\_url]

links.add(target\_id)

if links: # Only include pages that have outgoing links

outgoing\_links[url\_id] = links

# Update incoming links

for target\_id in links:

incoming\_links[target\_id].add(url\_id)

# Identify active pages (those with links)

active\_pages = set(outgoing\_links.keys()) | set(incoming\_links.keys())

# Initialize hub and authority scores only for active pages

hub\_scores = {url\_id: 1.0 for url\_id in active\_pages}

auth\_scores = {url\_id: 1.0 for url\_id in active\_pages}

# HITS algorithm iterations

EPSILON = 1e-10 # Minimum threshold for normalization

for iteration in range(max\_iterations):

old\_hub\_scores = hub\_scores.copy()

old\_auth\_scores = auth\_scores.copy()

# Update authority scores

new\_auth = {}

for page in active\_pages:

new\_auth[page] = sum(hub\_scores[incoming] for incoming in incoming\_links[page])

# Normalize authority scores

auth\_norm = math.sqrt(sum(score \* score for score in new\_auth.values()))

if auth\_norm < EPSILON:

auth\_norm = EPSILON

for page in new\_auth:

new\_auth[page] /= auth\_norm

auth\_scores = new\_auth

# Update hub scores

new\_hub = {}

for page in active\_pages:

new\_hub[page] = sum(auth\_scores[outgoing] for outgoing in outgoing\_links[page])

# Normalize hub scores

hub\_norm = math.sqrt(sum(score \* score for score in new\_hub.values()))

if hub\_norm < EPSILON:

hub\_norm = EPSILON

for page in new\_hub:

new\_hub[page] /= hub\_norm

hub\_scores = new\_hub

# Check convergence using maximum difference

max\_diff = max(

max(abs(auth\_scores[p] - old\_auth\_scores[p]) for p in active\_pages),

max(abs(hub\_scores[p] - old\_hub\_scores[p]) for p in active\_pages)

)

if max\_diff < epsilon:

print(f"Converged after {iteration + 1} iterations with max difference: {max\_diff:.6e}")

break

# Print analysis results

print("\n=== HITS Algorithm Analysis ===")

print("\nTop Hubs (pages that point to good authorities):")

sorted\_hubs = sorted(hub\_scores.items(), key=lambda x: x[1], reverse=True)

for url\_id, score in sorted\_hubs[:5]:

print(f"URL ID: {url\_id}, URL: {self.id\_to\_url[url\_id]}")

print(f"Hub Score: {score:.6f}")

print(f"Number of Outgoing Links: {len(outgoing\_links[url\_id])}")

print("\nTop Authorities (pages pointed to by good hubs):")

sorted\_authorities = sorted(auth\_scores.items(), key=lambda x: x[1], reverse=True)

for url\_id, score in sorted\_authorities[:5]:

print(f"URL ID: {url\_id}, URL: {self.id\_to\_url[url\_id]}")

print(f"Authority Score: {score:.6f}")

print(f"Number of Incoming Links: {len(incoming\_links[url\_id])}")

return {

'hub\_scores': hub\_scores,

'auth\_scores': auth\_scores,

'outgoing\_links': outgoing\_links,

'incoming\_links': incoming\_links

}

def get\_url\_id(self, url):

"""

Retrieves the unique integer ID for a given URL, assigning a new one if necessary.

Args:

url (str): The URL to look up or assign an ID to.

Returns:

int: The unique integer ID corresponding to the given URL.

"""

if url not in self.url\_to\_id:

self.url\_to\_id[url] = self.next\_id

self.id\_to\_url[self.next\_id] = url

self.next\_id += 1

return self.url\_to\_id[url]

def fetch\_page(self, url, max\_retries=3):

"""

Fetches page content with retry logic for 404 errors.

Args:

url (str): URL to fetch

max\_retries (int): Maximum retry attempts

Returns:

BeautifulSoup or None

"""

retries = 0

while retries < max\_retries:

try:

time.sleep(1)

response = requests.get(url, timeout=10)

if response.status\_code == 200:

return BeautifulSoup(response.text, 'html.parser')

elif response.status\_code == 404:

print(f"404 error for {url}, attempt {retries + 1}/{max\_retries}")

retries += 1

if retries < max\_retries:

time.sleep(2) # Wait before retrying

continue

else:

print(f"Failed to fetch {url}: Status code {response.status\_code}")

return None

except Exception as e:

print(f"Error fetching {url}: {str(e)}")

return None

print(f"Failed to fetch {url} after {max\_retries} attempts")

return None

def get\_links(self, soup, base\_url):

"""

Extracts all <a> tag hyperlinks from a BeautifulSoup page and converts them to absolute URLs.

Args:

soup (BeautifulSoup): The parsed HTML content.

base\_url (str): The base URL to resolve relative links.

Returns:

set: A filtered set of absolute URLs found in the page.

"""

links = set()

if not soup:

return links

excluded\_patterns = [

'/contact',

'contact-form',

'kontakt',

'impressum',

'imprint',

'privacy',

'datenschutz',

'disclaimer',

'login',

'register',

'signin',

'signup',

'search',

'suche',

'/index',

'print',

'rss',

'feed',

'sitemap',

'archive',

'newsletter',

'tasks-and-organisation',

'user-information',

'index.html',

'#c',

'release',

'press-release',

'article\_list',

'press-contact',

]

excluded\_extensions = [

'.pdf', '.doc', '.docx', '.xls', '.xlsx',

'.zip', '.rar', '.mp3', '.mp4', '.avi',

'.jpg', '.jpeg', '.png', '.gif'

]

for link in soup.find\_all('a', href=True):

url = link['href']

absolute\_url = urljoin(base\_url, url)

if absolute\_url.startswith(base\_url):

# מחלץ את החלק שאחרי ה-base\_url

path = absolute\_url[len(base\_url):]

# בודק אם החלק שאחרי ה-base\_url מכיל תבניות אסורות

if not any(pattern in path.lower() for pattern in excluded\_patterns):

# בודק סיומות אסורות

if not any(absolute\_url.lower().endswith(ext) for ext in excluded\_extensions):

links.add(absolute\_url)

return links

def index\_words(self, soup, url):

"""

Extracts and indexes the words from a BeautifulSoup page, updating the

inverted and global indexes with stemmed words.

Args:

soup (BeautifulSoup): Parsed HTML content.

url (str): The URL of the page being indexed.

Returns:

dict: A dictionary of {stemmed\_word: count\_in\_this\_document} for the page.

"""

index = defaultdict(int)

url\_id = self.get\_url\_id(url)

if soup:

words = re.findall(r'\w+', soup.get\_text())

for word in words:

word = word.lower()

stemmed\_word = self.stemmer.stem(word)

# נבדוק אם מותר להשתמש במילה (לא מספר, לא קצר מדי וכד')

if not self.is\_valid\_word(stemmed\_word):

continue

if stemmed\_word not in self.stop\_words:

if stemmed\_word == "germani":

stemmed\_word="german"

# Increase counters in the local index (for this page)

index[stemmed\_word] += 1

# Update the global/inverted index

self.inverted\_index[stemmed\_word][url\_id] += 1

self.global\_index[stemmed\_word][url] += 1

return dict(index)

def crawl\_and\_index(self, start\_url, query, max\_pages=51):

"""

Performs focused crawling based on query relevance while maintaining soup cache for PageRank.

Args:

start\_url (str): Initial URL

query (str): Search query

max\_pages (int): Maximum pages to crawl

Returns:

tuple: (page\_indexes, soup\_cache)

"""

query\_words = re.findall(r'\w+', query.lower())

query\_terms = {self.stemmer.stem(word) for word in query\_words

if word not in self.stop\_words}

#visited = set()

#to\_visit = {start\_url}

visited = set([start\_url]) # Add start\_url to visited immediately

links = self.get\_links(self.fetch\_page(start\_url), start\_url)

to\_visit = set(links) # Initialize with links from start page

page\_indexes = {}

potential\_pages = []

seen\_content = set()

soup\_cache = {}

while to\_visit and len(visited) < 100:

url = to\_visit.pop()

if url in visited:

continue

print(f"Crawling: {url} (ID: {self.get\_url\_id(url)})")

soup = self.fetch\_page(url)

if soup:

# Cache the soup object

soup\_cache[url] = soup

title = soup.find('title')

headers = soup.find\_all(['h1', 'h2', 'h3'])

title\_terms = set()

if title:

title\_words = re.findall(r'\w+', title.text.lower())

title\_terms.update(self.stemmer.stem(w) for w in title\_words

if w not in self.stop\_words)

for h in headers:

header\_words = re.findall(r'\w+', h.text.lower())

title\_terms.update(self.stemmer.stem(w) for w in header\_words

if w not in self.stop\_words)

page\_index = self.index\_words(soup, url)

page\_terms = set(page\_index.keys())

# Always index the page for PageRank analysis

doc\_id = self.get\_url\_id(url)

page\_indexes[doc\_id] = page\_index

# Calculate relevance score if query terms exist

if page\_terms & query\_terms:

total\_words = sum(page\_index.values())

content = frozenset(page\_index.items())

if content not in seen\_content:

seen\_content.add(content)

page\_score = 0

for term in query\_terms:

if term in page\_index:

tf = page\_index[term] / total\_words

if term in title\_terms:

tf \*= 1.5

df = len(self.inverted\_index[term])

idf = math.log(len(self.url\_to\_id) / (df + 1))

page\_score += tf \* idf

norm = math.sqrt(page\_score \*\* 2)

if norm > 0:

page\_score /= norm

potential\_pages.append((doc\_id, page\_index, page\_score))

visited.add(url)

links = self.get\_links(soup, start\_url)

to\_visit.update(links - visited)

# Sort pages by relevance score

if potential\_pages:

top\_pages = sorted(potential\_pages, key=lambda x: x[2], reverse=True)[:50]

print("\nTop 50 pages by relevance score:")

for doc\_id, page\_index, score in top\_pages:

url\_str = self.id\_to\_url.get(doc\_id, "N/A")

print(f"Page ID: {doc\_id}, URL: {url\_str}")

relevant\_indexes = {doc\_id: index for doc\_id, index, \_ in top\_pages}

return relevant\_indexes, soup\_cache

else:

# אם לא נמצאו עמודים רלוונטיים – מיון על פי כמות התוכן (סכום ספירות המילים)

sorted\_page\_indexes = dict(sorted(page\_indexes.items(),

key=lambda item: sum(item[1].values()),

reverse=True))[:50]

print("\nTop 50 pages by content size:")

for doc\_id, page\_index in sorted\_pages:

url\_str = self.id\_to\_url.get(doc\_id, "N/A")

total\_words = sum(page\_index.values())

print(f"Page ID: {doc\_id}, URL: {url\_str}")

return sorted\_page\_indexes, soup\_cache

def is\_english\_word(self, word):

"""

בודק אם המילה מכילה רק תווים באנגלית

"""

return all(ord('a') <= ord(c.lower()) <= ord('z') for c in word)

def is\_english\_word(word):

"""

Checks if a word contains only English characters.

Returns True if the word only contains ASCII letters a-z or A-Z.

"""

return all(ord('a') <= ord(c.lower()) <= ord('z') for c in word if c.isalpha())

def compute\_inverted\_index(self, page\_indexes):

"""

מחשב inverted index עם נורמליזציית L2, כולל סינון מילות עצירה

"""

# אימון מודל Word2Vec על המסמכים - סינון מילות עצירה כבר בשלב זה

sentences = []

for doc\_id, word\_counts in page\_indexes.items():

sentence = []

for word, count in word\_counts.items():

# בדיקה שהמילה אינה מילת עצירה לפני הוספתה

if word not in self.stop\_words:

sentence.extend([word] \* count)

sentences.append(sentence)

model = Word2Vec(sentences, vector\_size=100, window=5, min\_count=1)

# חישוב חשיבות המילים עם TF-IDF

word\_importance = defaultdict(float)

N = len(page\_indexes) # מספר המסמכים

# חישוב TF-IDF vectors לכל מסמך

doc\_vectors = {}

for doc\_id, word\_counts in page\_indexes.items():

vector = defaultdict(float)

# חישוב סך המילים ללא מילות עצירה

total\_words = sum(count for word, count in word\_counts.items()

if word not in self.stop\_words)

for word, count in word\_counts.items():

# התעלמות ממילות עצירה

if word not in self.stop\_words and word in model.wv:

# חישוב TF

tf = count / total\_words if total\_words > 0 else 0

# חישוב IDF (רק עבור מסמכים עם המילה שאינה מילת עצירה)

df = len([1 for doc in page\_indexes.values()

if word in doc and word not in self.stop\_words])

idf = math.log(N / df) if df > 0 else 0

# TF-IDF score

vector[word] = tf \* idf

# L2 normalization

magnitude = math.sqrt(sum(score \* score for score in vector.values()))

if magnitude > 0:

for word in vector:

vector[word] /= magnitude

doc\_vectors[doc\_id] = vector

# חישוב חשיבות סופית של המילים

for word in model.wv.index\_to\_key:

if word not in self.stop\_words: # בדיקה נוספת למילות עצירה

word\_score = 0

for doc\_id, vector in doc\_vectors.items():

if word in vector:

# הוספת ציון TF-IDF המנורמל

word\_score += vector[word]

# חישוב דמיון סמנטי

try:

similar\_words = model.wv.most\_similar(word, topn=5)

# סינון מילים דומות שהן מילות עצירה

similar\_words = [(w, s) for w, s in similar\_words

if w not in self.stop\_words]

if similar\_words:

semantic\_score = sum(score for \_, score in similar\_words) / len(similar\_words)

word\_score \*= (1 + semantic\_score)

except KeyError:

continue

word\_importance[word] = word\_score

# בחירת 15 המילים המשמעותיות ביותר (שאינן מילות עצירה)

top\_15\_words = sorted(word\_importance.items(), key=lambda x: x[1], reverse=True)[:15]

top\_15\_set = {word for word, \_ in top\_15\_words}

# יצירת inverted index חדש רק עם 15 המילים הנבחרות

new\_inverted\_index = defaultdict(lambda: defaultdict(int))

for word in top\_15\_set:

for doc\_id, vector in doc\_vectors.items():

if word in vector:

new\_inverted\_index[word][doc\_id] = vector[word] # שמירת הערך המנורמל

return new\_inverted\_index, top\_15\_words

def save\_to\_excel(self, page\_indexes, filename="results.xlsx"):

"""

שומרת את תוצאות הניתוח לקובץ Excel

"""

# שימוש ב-compute\_inverted\_index לקבלת ה-inverted index המעודכן וה-model

new\_inverted\_index, top\_15\_words = self.compute\_inverted\_index(page\_indexes)

self.inverted\_index = new\_inverted\_index

# יצירת מודל Word2Vec חדש (נחוץ לחישוב מילים דומות)

sentences = []

for doc\_id, word\_counts in page\_indexes.items():

sentence = []

for word, count in word\_counts.items():

sentence.extend([word] \* count)

sentences.append(sentence)

model = Word2Vec(sentences, vector\_size=100, window=5, min\_count=1)

# יצירת DataFrame למיפוי URL-ID

url\_mappings = [{"URL ID": url\_id, "URL": url}

for url, url\_id in self.url\_to\_id.items()]

url\_df = pd.DataFrame(url\_mappings)

# יצירת DataFrame לספירות מילים בכל דף

word\_count\_rows = []

for url\_id, index in page\_indexes.items():

for word, count in index.items():

word\_count\_rows.append({

"URL ID": url\_id,

"Word": word,

"Count": count

})

counts\_df = pd.DataFrame(word\_count\_rows)

# יצירת DataFrame ל-inverted index

inverted\_rows = []

for word, importance in top\_15\_words:

doc\_counts = self.inverted\_index[word]

document\_ids = ", ".join(map(str, sorted(doc\_counts.keys())))

doc\_count = len(doc\_counts)

total\_count = sum(doc\_counts.values())

avg\_count = total\_count / doc\_count if doc\_count > 0 else 0

# חישוב מילים דומות באמצעות Word2Vec

similar\_words = "N/A"

if word in model.wv:

try:

similar = model.wv.most\_similar(word, topn=3)

similar\_words = ", ".join(f"{w}({s:.2f})" for w, s in similar)

except KeyError:

pass

inverted\_rows.append({

"Word": word,

"Document IDs": document\_ids,

"Number of Documents": doc\_count,

"Total Occurrences": total\_count,

"Average Count per Document": f"{avg\_count:.2f}",

"Word2Vec Score": f"{importance:.4f}",

"Similar Words": similar\_words

})

inverted\_df = pd.DataFrame(inverted\_rows)

# שמירה לקובץ Excel

with pd.ExcelWriter(filename, engine="openpyxl") as writer:

url\_df.to\_excel(writer, index=False, sheet\_name="URL Mappings")

counts\_df = counts\_df.sort\_values(by="URL ID")

counts\_df.reset\_index(drop=True, inplace=True)

counts\_df.to\_excel(writer, index=False, sheet\_name="Page Word Counts")

inverted\_df.to\_excel(writer, index=False, sheet\_name="Top 15 Inverted Index")

# הדפסת סיכום

print(f"\nResults saved to '{filename}'.")

print("\nTop 15 Words with Word2Vec Analysis:")

print("-" \* 60)

for word, importance in top\_15\_words:

doc\_count = len(self.inverted\_index[word])

total\_count = sum(self.inverted\_index[word].values())

print(f"Word: {word}")

print(f"- Importance Score: {importance:.4f}")

print(f"- Appears in {doc\_count} documents")

print(f"- Total occurrences: {total\_count}")

print("-" \* 30)

def calculate\_tf\_idf\_for\_query(self, query, page\_indexes, filename="results.xlsx"):

"""

Calculates TF-IDF scores for each query word in each document, applies L2 normalization,

and saves the results to an Excel sheet.

Steps:

1) Tokenize, remove stopwords, and stem the query.

2) For each document, compute TF (term frequency), IDF (inverse document frequency),

and TF-IDF for each query word.

3) Apply L2 normalization to TF-IDF scores.

4) Save results in a new sheet named "Query TF-IDF".

Args:

query (str): The user query.

page\_indexes (dict): Dictionary of {doc\_id: {word: count}} representing indexed documents.

filename (str): Name of the Excel file to append the TF-IDF results.

Returns:

None

"""

# 1) Preprocess the query

query\_words = re.findall(r'\w+', query.lower())

processed\_query = []

for qw in query\_words:

if qw not in self.stop\_words:

stemmed\_qw = self.stemmer.stem(qw)

processed\_query.append(stemmed\_qw)

# 2) Collect stats for TF-IDF

N = len(page\_indexes) # total number of documents

doc\_total\_words = {}

for doc\_id, index\_dict in page\_indexes.items():

total\_count = sum(index\_dict.values())

doc\_total\_words[doc\_id] = total\_count

# 3) Compute TF, IDF, TF-IDF (before normalization)

# First, collect all TF-IDF values grouped by document

doc\_tf\_idfs = defaultdict(list) # {doc\_id: [tf\_idf values]}

tf\_idf\_data = [] # Store all the data for later use

for doc\_id, index\_dict in page\_indexes.items():

url = self.id\_to\_url[doc\_id]

total\_words\_in\_doc = doc\_total\_words[doc\_id]

for qw in processed\_query:

# Calculate TF

freq\_in\_doc = self.inverted\_index[qw].get(doc\_id, 0)

tf = freq\_in\_doc / total\_words\_in\_doc if total\_words\_in\_doc > 0 else 0.0

# Calculate IDF

df = len(self.inverted\_index[qw].keys()) # number of docs containing qw

idf = math.log(N / df, 10) if df > 0 else 0.0

if df > 0:

raw\_idf = math.log(N / df, 10)

idf = max(raw\_idf, 0)

else:

idf = 0

# Calculate TF-IDF

tf\_idf = tf \* idf

# Store TF-IDF for later normalization

doc\_tf\_idfs[doc\_id].append(tf\_idf)

# Store all data for final output

tf\_idf\_data.append({

"doc\_id": doc\_id,

"url": url,

"query\_word": qw,

"tf": tf,

"idf": idf,

"tf\_idf": tf\_idf

})

# 4) Compute L2 norms for each document

doc\_norms = {}

for doc\_id, tf\_idfs in doc\_tf\_idfs.items():

l2\_norm = math.sqrt(sum(x \* x for x in tf\_idfs))

doc\_norms[doc\_id] = l2\_norm if l2\_norm > 0 else 1.0 # Avoid division by zero

# 5) Create final results with normalized values

tf\_idf\_rows = []

for data in tf\_idf\_data:

doc\_id = data["doc\_id"]

# Apply L2 normalization

normalized\_tf\_idf = data["tf\_idf"] / doc\_norms[doc\_id]

tf\_idf\_rows.append({

"Document ID": doc\_id,

"URL": data["url"],

"Query Concept": data["query\_word"],

"TF": data["tf"],

"IDF": data["idf"],

"TF-IDF": data["tf\_idf"],

"Normalized TF-IDF": normalized\_tf\_idf

})

# 6) Create DataFrame and save to Excel

tf\_idf\_df = pd.DataFrame(tf\_idf\_rows)

# Sort by Document ID and Normalized TF-IDF for better readability

tf\_idf\_df = tf\_idf\_df.sort\_values(

by=['Document ID', 'Normalized TF-IDF'],

ascending=[True, False]

)

try:

with pd.ExcelWriter(filename, engine="openpyxl", mode="a", if\_sheet\_exists="replace") as writer:

tf\_idf\_df.to\_excel(writer, index=False, sheet\_name="Query TF-IDF")

print(f"TF-IDF results (including L2 normalization) saved to '{filename}'.")

except FileNotFoundError:

with pd.ExcelWriter(filename, engine="openpyxl") as writer:

tf\_idf\_df.to\_excel(writer, index=False, sheet\_name="Query TF-IDF")

print(f"File '{filename}' was created and TF-IDF results (including L2 normalization) were saved.")

def select\_connected\_pages(page\_indexes, url\_to\_id, id\_to\_url, soup\_cache=None, max\_pages=50):

"""

Selects up to `max\_pages` pages from the crawled set that are interconnected

(i.e., have links among themselves), based on the stored soups.

Args:

page\_indexes (dict): Dictionary of {doc\_id: {word: count}}, currently not used

but included for future expansion.

url\_to\_id (dict): Mapping from URL to its assigned integer ID.

id\_to\_url (dict): Mapping from integer ID to the original URL string.

soup\_cache (dict): A dictionary mapping URL -> BeautifulSoup object (for faster access).

max\_pages (int): Maximum number of pages to select.

Returns:

dict: A dictionary of {page\_id: [list\_of\_outgoing\_page\_ids]} for the selected connected pages.

"""

connected\_pages = {}

page\_links = {}

# Build initial link structure

for url\_id in id\_to\_url:

url = id\_to\_url[url\_id]

soup = soup\_cache.get(url) if soup\_cache else None

if soup:

links = set()

for link in soup.find\_all('a', href=True):

absolute\_url = urljoin(url, link['href'])

if absolute\_url in url\_to\_id:

target\_id = url\_to\_id[absolute\_url]

links.add(target\_id)

if links: # Only include pages that have outgoing links

page\_links[url\_id] = list(links)

# Select connected pages (prefer pages with more connections)

selected\_ids = set()

for page\_id, links in sorted(page\_links.items(), key=lambda x: len(x[1]), reverse=True):

if len(selected\_ids) >= max\_pages:

break

connected\_links = [l for l in links if l in page\_links]

if connected\_links:

selected\_ids.add(page\_id)

selected\_ids.update(connected\_links[:max\_pages-len(selected\_ids)])

# Create final link structure for selected pages

for page\_id in selected\_ids:

outgoing = [l for l in page\_links.get(page\_id, []) if l in selected\_ids]

if outgoing:

connected\_pages[page\_id] = outgoing

return connected\_pages

def calculate\_new\_pagerank(current\_ranks, links, damping\_factor=0.85, epsilon=1e-6):

"""

Computes new PageRank values for one iteration with a damping factor,

handling dangling nodes, and includes a convergence check.

Args:

current\_ranks (dict): Current PageRank scores {page\_id: score}.

links (dict): Adjacency list {page\_id: [list\_of\_outgoing\_ids]}.

damping\_factor (float): Damping factor for PageRank (usually around 0.85).

epsilon (float): Threshold for convergence.

Returns:

tuple:

new\_ranks (dict): Updated PageRank scores {page\_id: score}.

converged (bool): True if the algorithm has converged under epsilon, else False.

"""

new\_ranks = {}

num\_pages = len(current\_ranks)

dangling\_sum = sum(current\_ranks[page] for page in current\_ranks if page not in links)

# Iterate over each page to calculate its new rank

for page in current\_ranks:

rank\_sum = 0

# Find incoming links to the page

incoming\_links = [p for p, outgoing in links.items() if page in outgoing]

# Sum up contributions from incoming links

for source\_page in incoming\_links:

num\_outgoing = len(links[source\_page])

if num\_outgoing > 0: # Avoid division by zero

rank\_sum += current\_ranks[source\_page] / num\_outgoing

# Add damping factor, dangling nodes contribution, and random jump

new\_ranks[page] = (

(1 - damping\_factor) / num\_pages + # Random jump

damping\_factor \* (rank\_sum + dangling\_sum / num\_pages) # Link contributions

)

# Check for convergence

max\_change = max(abs(new\_ranks[page] - current\_ranks[page]) for page in current\_ranks)

if max\_change < epsilon:

print(f"Converged with max change: {max\_change:.6e}")

return new\_ranks, True # Return new ranks and a flag indicating convergence

return new\_ranks, False # Return new ranks and a flag indicating non-convergence

def print\_ranks(ranks, iteration, id\_to\_url=None):

"""

Prints PageRank values after a given iteration.

Args:

ranks (dict): Current PageRank scores {page\_id: score}.

iteration (int): The iteration number.

id\_to\_url (dict): Mapping from page\_id to URL; if provided, the URL is displayed too.

Returns:

None

"""

print(f"\nPageRank values after iteration {iteration}:")

print("-" \* 50)

if id\_to\_url:

print("Page ID | URL | PageRank Value")

else:

print("Page | PageRank Value")

print("-" \* 50)

for page, rank in ranks.items():

if id\_to\_url:

url = id\_to\_url[page]

print(f"{page} | {url[:100]}... | {rank:.3f}")

else:

print(f"{page} | {rank:.3f}")

print("-" \* 50)

def run\_pagerank\_analysis(page\_indexes, url\_to\_id, id\_to\_url, soup\_cache, num\_iterations=2):

"""

Executes a full PageRank computation workflow:

1) Selects connected pages.

2) Prints the initial PageRank values.

3) Iteratively computes new PageRank values (up to num\_iterations or convergence).

4) Identifies the page with the highest final PageRank score.

Args:

page\_indexes (dict): Dictionary of {doc\_id: {word: count}} from the crawl.

url\_to\_id (dict): Mapping from URL to integer IDs.

id\_to\_url (dict): Mapping from integer IDs to URL.

soup\_cache (dict): A cache mapping URL -> BeautifulSoup objects for faster access.

num\_iterations (int): Maximum number of PageRank iterations.

Returns:

dict: Final PageRank scores {page\_id: score}.

"""

print("\n=== PageRank Analysis ===")

# Select connected pages

connected\_pages = select\_connected\_pages(page\_indexes, url\_to\_id, id\_to\_url, soup\_cache)

if not connected\_pages:

print("No connected pages found. PageRank analysis cannot proceed.")

return

# Show selected pages

print("\nSelected pages for PageRank analysis:")

for page\_id in connected\_pages:

print(f"\nPage ID: {page\_id}, URL: {id\_to\_url[page\_id]}")

print(f"Links to: {[id\_to\_url[link\_id] for link\_id in connected\_pages[page\_id]]}")

# Initialize PageRank values

current\_ranks = {page: 1/len(connected\_pages) for page in connected\_pages}

# Print initial values

print("\nInitial PageRank Values:")

print\_ranks(current\_ranks, 0, id\_to\_url)

# Perform iterations

for i in range(num\_iterations):

# current\_ranks = calculate\_new\_pagerank(current\_ranks, connected\_pages)

# print\_ranks(current\_ranks, i + 1, id\_to\_url)

current\_ranks, converged = calculate\_new\_pagerank(current\_ranks, connected\_pages)

print\_ranks(current\_ranks, i + 1, id\_to\_url)

#print(f"Iteration {i+1}: {current\_ranks}")

if converged:

break

# Find highest PageRank

highest\_page = max(current\_ranks.items(), key=lambda x: x[1])

print(f"\n\nHighest PageRank after {num\_iterations} iterations:")

print(f"Page ID: {highest\_page[0]}")

print(f"URL: {id\_to\_url[highest\_page[0]]}")

print(f"PageRank value: {highest\_page[1]:.3f}")

return current\_ranks

class FeedbackRelevance:

def \_\_init\_\_(self, pages, pagerank\_scores, id\_to\_url, search\_engine):

"""

Manages a user feedback system for re-ranking results using relevance feedback.

Args:

pages (list): List of page IDs to present to users.

pagerank\_scores (dict): PageRank scores {page\_id: score}.

id\_to\_url (dict): Mapping from page\_id to URL.

search\_engine (SearchEngine): An instance of the SearchEngine class.

Attributes:

pages (list): The pages to present for feedback.

pagerank\_scores (dict): Current PageRank scores for each page.

id\_to\_url (dict): Map from page\_id to URL.

search\_engine (SearchEngine): Reference to the main SearchEngine instance.

user\_feedback (dict): Stores feedback per user (e.g., {user\_id: {page\_id: bool}}).

outputs (dict): Stores output widgets per user for displaying status/results.

button\_pairs (dict): Stores pairs of (relevant\_button, non\_relevant\_button) per user.

"""

self.pages = pages # List of page IDs

self.pagerank\_scores = pagerank\_scores

self.id\_to\_url = id\_to\_url

self.search\_engine = search\_engine # Reference to SearchEngine instance

self.user\_feedback = {} # Dictionary to store feedback per user

self.outputs = {} # Dictionary to store outputs per user

self.button\_pairs = {} # Dictionary to store button pairs

def initialize\_user(self, user\_id):

"""

Ensures internal data structures are set up for a given user.

Args:

user\_id (int): Unique identifier for the user.

Returns:

None

"""

if user\_id not in self.user\_feedback:

self.user\_feedback[user\_id] = {}

self.outputs[user\_id] = widgets.Output()

self.button\_pairs[user\_id] = {}

def create\_page\_display(self, page\_id):

"""

Builds an HTML snippet summarizing a page (URL, headings, etc.) for display.

Args:

page\_id (int): The unique ID for the page to display.

Returns:

str: An HTML string with page information.

"""

url = self.id\_to\_url[page\_id]

# Get the page soup from search engine's cache

soup = self.search\_engine.fetch\_page(url) if hasattr(self.search\_engine, 'fetch\_page') else None

# Extract headings

headings = []

if soup:

# Get main headings (h1, h2)

for h in soup.find\_all(['h1', 'h2'])[:2]:

text = h.get\_text().strip()

if text:

headings.append(text)

# Get the keywords from the search engine's index

words = self.search\_engine.global\_index.get(url, {}).keys()

page\_text = " ".join(words)

return f"""

<div style="margin: 20px; padding: 10px; border: 1px solid #ddd; border-radius: 5px;">

<div style="float: right; color: #666;">PageRank Score: {self.pagerank\_scores[page\_id]:.3f}</div>

<div style="color: #1a0dab; font-size: 18px;">Page ID: {page\_id}</div>

<div style="color: #006621; font-size: 14px;"><a href="{url}" target="\_blank">{url}</a></div>

<div style="color: #333; font-weight: bold; margin-top: 10px;">

{' | '.join(headings) if headings else 'No headings available'}

</div>

</div>

"""

def on\_button\_click(self, b):

"""

Handler for marking a document as relevant or non-relevant.

Updates internal state and changes button styles accordingly.

Args:

b (Button): The button that was clicked.

It contains 'user\_id' and 'page\_id' as custom attributes.

Returns:

None

"""

user\_id = int(b.user\_id)

page\_id = int(b.page\_id)

is\_relevant = b.description.startswith('👍')

# Update feedback

self.user\_feedback[user\_id][page\_id] = is\_relevant

# Update button styles

rel\_btn, nonrel\_btn = self.button\_pairs[user\_id][page\_id]

if is\_relevant:

rel\_btn.button\_style = 'success'

nonrel\_btn.button\_style = ''

else:

rel\_btn.button\_style = ''

nonrel\_btn.button\_style = 'danger'

# Show current feedback status

with self.outputs[user\_id]:

clear\_output()

print(f"✓ Marked document {page\_id} as {'relevant' if is\_relevant else 'not relevant'}")

if self.user\_feedback[user\_id]:

print("\nCurrent feedback:")

for pid, v in sorted(self.user\_feedback[user\_id].items()):

status = "relevant" if v else "not relevant"

print(f"Document {pid}: {status}")

def rerank\_results(self, user\_id):

"""

Re-ranks the pages using Rocchio-like relevance feedback.

Creates TF-IDF vectors for all pages, then adjusts the query vector

based on user feedback and computes new rankings via cosine similarity.

Args:

user\_id (int): Identifier for the user whose feedback should be used.

Returns:

tuple:

reranked\_pages (list): A list of page IDs in descending order of new scores.

scores (numpy.ndarray): Corresponding similarity scores for each page.

"""

relevant\_pages = [i for i, v in self.user\_feedback[user\_id].items() if v == True]

non\_relevant\_pages = [i for i, v in self.user\_feedback[user\_id].items() if v == False]

# Create TF-IDF vectors

all\_words = set()

page\_word\_counts = {}

# Calculate document frequencies (DF)

doc\_freq = defaultdict(int)

for page\_id in self.pages:

word\_counts = {word: counts[page\_id] for word, counts in self.search\_engine.inverted\_index.items()

if page\_id in counts}

for word in word\_counts:

doc\_freq[word] += 1

all\_words.update(word\_counts.keys())

page\_word\_counts[page\_id] = word\_counts

# Calculate total words per document

doc\_total\_words = {}

for doc\_id, word\_counts in page\_word\_counts.items():

doc\_total\_words[doc\_id] = sum(word\_counts.values())

# Create word-to-index mapping

word\_to\_idx = {word: idx for idx, word in enumerate(sorted(all\_words))}

# Create TF-IDF vectors

doc\_vectors = []

N = len(self.pages) # Total number of documents

for page\_id in self.pages:

vector = np.zeros(len(all\_words))

word\_counts = page\_word\_counts[page\_id]

total\_words = doc\_total\_words[page\_id]

for word, count in word\_counts.items():

idx = word\_to\_idx[word]

tf = count / total\_words if total\_words > 0 else 0

idf = math.log(N / doc\_freq[word]) if doc\_freq[word] > 0 else 0

vector[idx] = tf \* idf

doc\_vectors.append(vector)

doc\_vectors = np.array(doc\_vectors)

# Initialize query vector as mean of all documents

#To get an initial query vector, we take the average of all the document vectors.

query\_vector = np.mean(doc\_vectors, axis=0)

# Modify query vector based on feedback

beta = 0.75 # Weight for relevant documents

gamma = 0.15 # Weight for non-relevant documents

if relevant\_pages:

relevant\_indices = [self.pages.index(page\_id) for page\_id in relevant\_pages]

query\_vector += beta \* np.mean(doc\_vectors[relevant\_indices], axis=0)

if non\_relevant\_pages:

non\_relevant\_indices = [self.pages.index(page\_id) for page\_id in non\_relevant\_pages]

query\_vector -= gamma \* np.mean(doc\_vectors[non\_relevant\_indices], axis=0)

# Calculate similarities using sklearn's cosine\_similarity

similarities = cosine\_similarity(doc\_vectors, query\_vector.reshape(1, -1)).flatten()

ranked\_indices = np.argsort(-similarities)

return [self.pages[i] for i in ranked\_indices], similarities[ranked\_indices]

def show\_ranking(self, b):

"""

Displays a new ranking based on user feedback.

Triggered by "Show Ranking" button click.

Args:

b (Button): The button widget that triggered this method.

Returns:

None

"""

user\_id = int(b.user\_id)

if not self.user\_feedback[user\_id]:

with self.outputs[user\_id]:

clear\_output()

print("Please mark at least one document as relevant or not relevant.")

return

reranked\_pages, scores = self.rerank\_results(user\_id)

with self.outputs[user\_id]:

clear\_output()

print(f"🔄 Reranked results for User {user\_id}:\n")

for page\_id, score in zip(reranked\_pages, scores):

display(HTML(f"""

<div style="margin: 20px; padding: 10px; border: 1px solid #ddd; border-radius: 5px;">

<div style="float: right; color: #666;">

New Score: {score:.3f} | Original PageRank: {self.pagerank\_scores[page\_id]:.3f}

</div>

<div style="color: #1a0dab; font-size: 18px;">Page ID: {page\_id}</div>

<div style="color: #006621; font-size: 14px;">{self.id\_to\_url[page\_id]}</div>

<div style="color: #545454; margin-top: 5px;">

Feedback: {

"Relevant ✓" if self.user\_feedback[user\_id].get(page\_id) == True

else "Not Relevant ✗" if self.user\_feedback[user\_id].get(page\_id) == False

else "No Feedback"

}

</div>

</div>

"""))

def display\_interface(self, user\_id):

"""

Builds and displays the interactive feedback interface (page listings,

relevant/not relevant buttons, and ranking buttons) for a specific user.

Args:

user\_id (int): Unique identifier for the user.

Returns:

None

"""

self.initialize\_user(user\_id)

# Create header

display(HTML(f"""

<div style="margin: 20px 0; padding: 10px; background-color: #f8f9fa; border-radius: 5px;">

<h3 style="margin: 0; color: #333;">Feedback Interface for User {user\_id}</h3>

<p style="margin: 10px 0 0 0; color: #666;">

Click 👍 or 👎 to mark pages as relevant or not relevant, then use the buttons below

to see reordered results or get query suggestions.

</p>

</div>

"""))

for page\_id in self.pages:

display(HTML(self.create\_page\_display(page\_id)))

# Create buttons

rel\_btn = widgets.Button(

description='👍 Relevant',

tooltip=str(page\_id),

layout=widgets.Layout(margin='5px')

)

nonrel\_btn = widgets.Button(

description='👎 Not Relevant',

tooltip=str(page\_id),

layout=widgets.Layout(margin='5px')

)

# Add user\_id and page\_id as attributes

rel\_btn.user\_id = str(user\_id)

rel\_btn.page\_id = str(page\_id)

nonrel\_btn.user\_id = str(user\_id)

nonrel\_btn.page\_id = str(page\_id)

# Store button pair

self.button\_pairs[user\_id][page\_id] = (rel\_btn, nonrel\_btn)

# Set up click handlers

rel\_btn.on\_click(self.on\_button\_click)

nonrel\_btn.on\_click(self.on\_button\_click)

display(widgets.HBox([rel\_btn, nonrel\_btn]))

# Create action buttons container

button\_container = widgets.HBox([

widgets.Button(

description='Show Ranking',

button\_style='primary',

layout=widgets.Layout(margin='20px 10px 20px 0')

),

widgets.Button(

description='Show Query Suggestion',

button\_style='info',

layout=widgets.Layout(margin='20px 0 20px 10px')

)

])

# Set up button attributes and handlers

rank\_button = button\_container.children[0]

suggest\_button = button\_container.children[1]

rank\_button.user\_id = str(user\_id)

suggest\_button.user\_id = str(user\_id)

rank\_button.on\_click(self.show\_ranking)

suggest\_button.on\_click(self.display\_query\_suggestion)

display(button\_container)

display(self.outputs[user\_id])

def suggest\_expanded\_query(self, user\_id):

"""

Generates additional query terms using relevance feedback from the user.

Calculates term weights, and selects top terms that appear in relevant documents

but are less frequent in non-relevant ones.

Args:

user\_id (int): Unique identifier for the user.

Returns:

tuple:

expanded\_query (str or None): Suggested query terms joined by spaces,

or None if no suggestion is possible.

explanations (str or None): Explanation of chosen terms, or None if not applicable.

"""

# Get relevant and non-relevant documents

relevant\_pages = [i for i, v in self.user\_feedback[user\_id].items() if v == True]

non\_relevant\_pages = [i for i, v in self.user\_feedback[user\_id].items() if v == False]

if not relevant\_pages:

return None, None

# Collect terms from relevant documents

term\_weights = defaultdict(float)

term\_counts = defaultdict(int)

# Calculate term frequencies in relevant documents

for page\_id in relevant\_pages:

url = self.id\_to\_url[page\_id]

# Get terms from the inverted index

for term, doc\_counts in self.search\_engine.inverted\_index.items():

if page\_id in doc\_counts:

term\_weights[term] += doc\_counts[page\_id]

term\_counts[term] += 1

# Penalize terms that appear in non-relevant documents

for page\_id in non\_relevant\_pages:

url = self.id\_to\_url[page\_id]

for term, doc\_counts in self.search\_engine.inverted\_index.items():

if page\_id in doc\_counts:

term\_weights[term] -= doc\_counts[page\_id] \* 0.5 # Penalty factor

# Calculate final term weights using Rocchio-like approach

alpha = 1.0 # Original query weight

beta = 0.75 # Relevant documents weight

gamma = 0.15 # Non-relevant documents weight

# Sort terms by weight and get top N terms

N = 5 # Number of terms to suggest

suggested\_terms = sorted(

[(term, weight) for term, weight in term\_weights.items()],

key=lambda x: x[1],

reverse=True

)[:N]

# Create expanded query suggestion

expansion\_terms = []

term\_explanations = []

for term, weight in suggested\_terms:

if weight > 0:

expansion\_terms.append(term)

explanation = (

f"'{term}' (weight: {weight:.2f}, "

f"appears in {term\_counts[term]} relevant documents)"

)

term\_explanations.append(explanation)

if not expansion\_terms:

return None, None

expanded\_query = " ".join(expansion\_terms)

explanations = "\n".join(f"- {exp}" for exp in term\_explanations)

return expanded\_query, explanations

def display\_query\_suggestion(self, b):

"""

Button callback that displays a query expansion suggestion to the user.

Args:

b (Button): The button widget that triggered this method.

Returns:

None

"""

user\_id = int(b.user\_id)

if not self.user\_feedback[user\_id]:

with self.outputs[user\_id]:

clear\_output()

print("Please provide feedback on some documents first.")

return

expanded\_query, explanations = self.suggest\_expanded\_query(user\_id)

with self.outputs[user\_id]:

clear\_output()

if expanded\_query:

print("📝 Suggested Query Expansion:")

print(f"\nExpanded Query: {expanded\_query}")

print("\nTerm Selection Explanation:")

print(explanations)

print("\nThis suggestion is based on the terms that appear frequently in")

print("documents you marked as relevant, while avoiding terms from")

print("non-relevant documents.")

else:

print("Unable to generate query suggestion.")

print("Please mark more documents as relevant/non-relevant.")

def add\_suggestion\_button(self):

"""

(Optional) Adds a standalone button to show query suggestions.

Can be used if we want to manually place a separate suggestion button.

Returns:

None

"""

suggest\_button = widgets.Button(

description='Show Query Suggestion',

button\_style='info',

layout=widgets.Layout(margin='20px 0')

)

suggest\_button.user\_id = str(user\_id)

suggest\_button.on\_click(self.display\_query\_suggestion)

display(suggest\_button)

def implement\_feedback\_relevance(pagerank\_scores, id\_to\_url, search\_engine):

"""

Builds and displays the feedback interface for multiple users

to provide relevance feedback on the pages.

Args:

pagerank\_scores (dict): PageRank scores {page\_id: score}.

id\_to\_url (dict): Mapping from page\_id to URL.

search\_engine (SearchEngine): Reference to the SearchEngine instance.

Returns:

FeedbackRelevance: The feedback system object to allow further interaction.

"""

# מיון הדפים לפי ציון PageRank וקבלת 20 המובילים

top\_20\_pages = sorted(pagerank\_scores.items(), key=lambda x: x[1], reverse=True)[:20]

# יצירת רשימת ה-IDs של 20 הדפים המובילים

top\_pages = [page\_id for page\_id, score in top\_20\_pages]

# יצירת מילון חדש עם הציונים רק של 20 הדפים הנבחרים

filtered\_scores = {page\_id: score for page\_id, score in top\_20\_pages}

# יצירת מערכת המשוב עם הדפים המסוננים

feedback\_system = FeedbackRelevance(top\_pages, filtered\_scores, id\_to\_url, search\_engine)

# הצגת הממשק לשני המשתמשים

print("\nDisplaying Feedback Interface for User 1...")

feedback\_system.display\_interface(1)

print("\nDisplaying Feedback Interface for User 2...")

feedback\_system.display\_interface(2)

return feedback\_system

def main():

"""

Main driver function with integrated crawling and PageRank analysis

Main driver function that:

1) Initializes the SearchEngine.

2) Performs crawling, indexing, and data caching.

3) Analyzes link structure (HITS).

4) Runs PageRank analysis.

5) Saves initial results to Excel.

6) Calculates TF-IDF for a sample query and appends it to an Excel file.

7) Initializes and displays a user feedback relevance system.

"""

search\_engine = SearchEngine()

start\_url = "https://www.bundesgesundheitsministerium.de/en/"

query = "What are the available healthcare policies for chronic diseases in Germany?"

print("Starting crawling and indexing...")

start\_crawl = time.time()

# Use crawl\_and\_index for both indexing and PageRank

page\_indexes, soup\_cache = search\_engine.crawl\_and\_index(start\_url, query,max\_pages=51)

end\_crawl = time.time()

crawl\_time = end\_crawl - start\_crawl

print(f"Crawling and indexing completed in {crawl\_time:.2f} seconds.")

print(f"Processing query: {query}")

start\_tfidf = time.time()

filename = f"results\_query\_{query}.xlsx"

search\_engine.calculate\_tf\_idf\_for\_query(query, page\_indexes, filename)

end\_tfidf = time.time()

tfidf\_time = end\_tfidf - start\_tfidf

with pd.ExcelWriter(filename, engine="openpyxl", mode="a", if\_sheet\_exists="replace") as writer:

timing\_data = pd.DataFrame([{

"Crawling Time (seconds)": crawl\_time,

"TF-IDF Calculation Time (seconds)": tfidf\_time

}])

timing\_data.to\_excel(writer, index=False, sheet\_name="Timing Information")

print(f"TF-IDF calculation completed in {tfidf\_time:.2f} seconds.")

print(f"Results saved to '{filename}'.")

# Save results and continue with existing functionality

search\_engine.save\_to\_excel(page\_indexes, "initial\_results.xlsx")

# Run HITS Algorithm Analysis

link\_analysis = search\_engine.analyze\_link\_structure(max\_iterations=20, epsilon=1e-8)

# Run PageRank analysis with the soup cache

pagerank\_scores = run\_pagerank\_analysis(

page\_indexes,

search\_engine.url\_to\_id,

search\_engine.id\_to\_url,

soup\_cache,

num\_iterations=2

)

# Initialize feedback system

print("\nInitializing Feedback Relevance System...")

feedback\_system = implement\_feedback\_relevance(pagerank\_scores, search\_engine.id\_to\_url, search\_engine)

if \_\_name\_\_ == "\_\_main\_\_":

main()