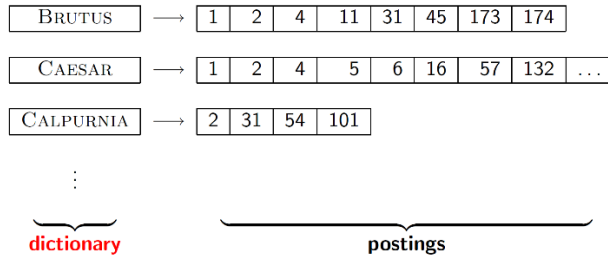


אחזור מידע

Dictionaries and tolerant retrieval

האינדקס (Inverted Index) בנוי ממילון מכיל רשימת ביטויים, מספר ההופעות של אותם הביטויים וכן מצביע ל- postings list וכמובן את ה- postings lists עצמם. מהו מבנה הנתונים בו אנו משתמשים עבור האינדקס? האם מבין האפשרויות השונות לייצוג האינדקס יש מבנה נתונים שהוא יעיל יותר?



אנו נתמקד כרגע במבנה של המילון. את המילון אפשר לשמור כמערך של-structים בעל המבנה הבא:

term	document frequency	pointer to postings list
a	656,265	→
aachen	65	→
...
zulu	221	→

כל רשומה בנויה מה-term, מס' המופעים שלו ומצביע ל-posting list המתאים. הרשומות עצמן ממוינות בהתאם ל-terms השונים. אם נניח שגודל ה-term הוא 20 בתים, מספר המופעים של ה-term (document frequency) הוא מסוג int (כלומר 4 בתים) וגם המצביע ל-posting list הוא מסוג (כלומר עוד 4 בתים), הרי שסה"כ כל רשומה בנויה מ-28 בתים. אנו יכולים למצוא כל term ע"י חיפוש בינארי. צורת המימוש הזאת היא הפשוטה ביותר. אנו מעוניינים במבנה נתונים יעיל יותר מבחינת צריכת הזיכרון, ושיאפשר לנו לבצע חיפוש יעיל (כלומר מהיר) במילון.

לצורך כך עומדות לרשותנו שתי אפשרויות עיקריות: (1) טבלאות Hash, ו-(2) עצי חיפוש.

טבלאות Hash

הערה: ההנחה היא שיש לכם ידע מוקדם בנושא ה-Hash וטבלאות Hash.

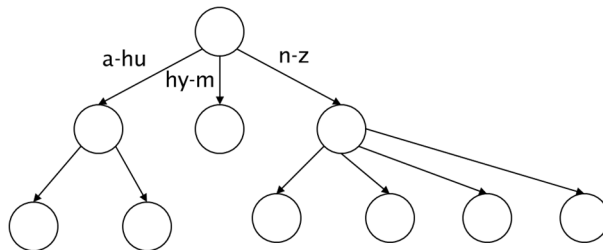
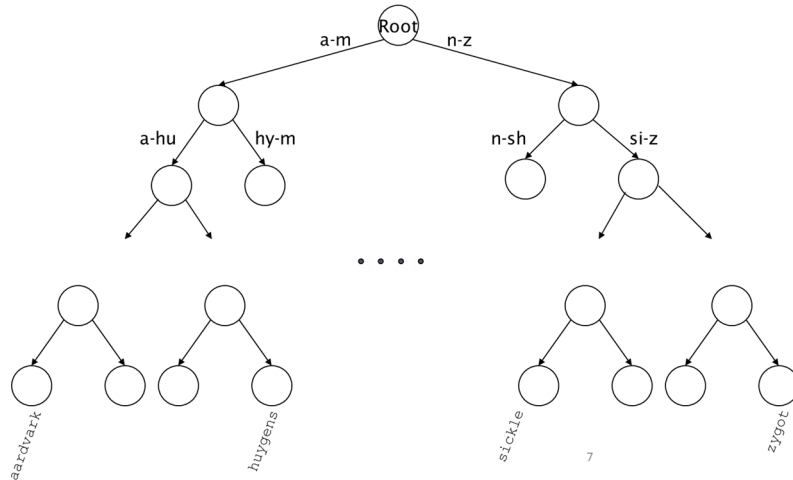
כל term במילון, ע"י שימוש בפונקציית ה-Hash מתורגם למספר (Integer). באופן זה אנו יכולים בצורה "מידית" לשמור כל רשומה (בהתאם ל-term) במקום ייחודי וידוע במערך של רשומות.

לשימוש בטבלאות Hash יתרון ברור. חיפוש של ביטוי בעזרת טבלאות ופונקציות Hash הינה פעולה בסדר גודל $O(1)$ (או קרוב $O(n)$, למשל בשימוש בדליים בגודל n).

אבל, ישנם גם מספר חסרונות לשימוש בטבלאות Hash. (1) יתכן ומס' terms ימופו לאותו הערך. במקרה כזה, יש לנו מספר שיטות לטפל בהתנגשויות הללו, אבל אז זמן החיפוש גדל במעט. (2) יתכן וביטויים דומים ימופו לערכים שונים. למשל judgment ו-judgement (כתיב אמריקאי/בריטי), לא בהכרח ימופו לאותו הערך ולכן קשה לקבל את שני המופעים (הם לא יופיעו אחד ליד השני בטבלה). (3) אי אפשר לבצע שאילתות כגון *judg. ו-(4) אם מספר הביטויים ממשך לגדול פונקציית ה-Hash תתחיל ליצור הרבה התנגשויות, במקרים כאלו יש צורך להחליף את פונקציית ה-Hash בכזו שיוצרת פחות התנגשויות, ובהתאם יש צורך (קבוע) לבצע Hashing מחדש לכל הביטויים בטבלה.

עצים

הגדרה כללית: מבנה נתונים בצורת עץ ששומר מידע ממוין ומאפשר חיפוש, גישה סדרתית, הוספת אברים ומחיקתם בסיבוכיות לוגריתמית. עץ בינארי הוא עץ, שבו לכל צומת יכולים להיות עד שני בנים. עץ B הוא הכללה של עץ חיפוש בינארי בכך שלכל צומת יכולים להיות יותר מ-2 בנים ובנוסף כל העלים באותו עומק. להבדיל מעצי חיפוש מאוזנים, עץ B מיועד לעבודה יעילה במערכות שקוראות וכותבות בלוקים גדולים של מידע.



עצי חיפוש הם עצים מאוזנים שמאפשרים חיפוש מהיר של איברים. קיימים מס' סוגים של עצים. הפשוטים הם עצים בינאריים. "מתוחכמים יותר", וגם שימושיים יותר הם עצי B (קיימים סוגים שונים של עצי B). היתרון הברור בשימוש בעצים הוא שעצים פותרים את בעיית התחליות.

אבל, בדומה לטבלאות Hash, גם לעצים יש חסרונות. כששני החסרונות העיקריים הם (1) חיפוש איטי יותר, $O(\log m)$, שתלוי בעומק העץ (ולכן, בכדי שיהיה יעיל דורש מאתנו עצים מאוזנים). ו-(2) תהליך האיזון מחדש, שנדרש כל פעם שמוסיפים term לעץ, הוא תהליך איטי. כאשר, בעצי B תהליך האיזון מחדש מתבצע באופן "אוטומטי" עקב המבנה המיוחד של העץ ואופי ביצוע פעולות ההוספה והמחיקה.

Wild-card queries

נניח שאנו רוצים לבצע את השאלתא הבאה: mon^* , כלומר למצוא את כל המסמכים בהם ישנם terms המתחילים mon-b.

אם אנו משתמשים בעצים בינאריים או עצי B, הפתרון הוא פשוט. נביא את כל ה-terms הנמצאים בטווח $mon \leq w < moo$, ועבור כל term נביא את רשימת המסמכים שבו הוא נמצא מה-position list שלו.

נניח שעכשיו אנו רוצים את השאלתא הבאה: mon^* , כלומר למצוא את כל המסמכים בהם ישנם terms המסתיימים mon-b. אנו לא יכולים לעשות זאת בעזרת העץ שיש לנו כרגע, מכיון שהחיפוש בו מבוסס על התחלות המילים.

אולם ניתן לפתור בעיה זו ע"י בנייה של עץ בינארי או עץ B נוסף, שבו terms כתובים מהסוף להתחלה. אם נשתמש בעץ הזה, השאילתא עבורו תשנה מ- mon^* ל- nom^* (מכוון שכל term בו כתוב מהסוף להתחלה).

שאילתא כגון, הבא את כל המושגים הנמצאים בטווח $non < w \leq nom$, דורשת עבודה עם שני העצים (הרגיל ו"ההפוך"), כשכל term שמתקבל, ונמצא בתוצאות של שני העצים, אנו נביא את רשימת המסמכים מה-position list שלו.

נניח ואנו רוצים למצוא את כל המסמכים בהם יש ביטויים המתחילים ב-co ומסתיימים ב- (co^*tion) . תחילה, נחפש את כל הביטויים המתחילים ב- (co^*) בעץ B, ועבור כל ביטוי ניקח את רשימת המסמכים שלו מה-posting index שלו. אח"כ, נחפש את כל הביטויים המסתיימים ב- $(^*tion)$ בעץ B "הפוך", ועבור כל ביטוי ניקח את רשימת המסמכים שלו מה-posting index שלו. לבסוף, נבצע פעולת AND בין התוצאות (כלומר אלו מסמכים מופיעים גם בחלק הראשון וגם בחלק השני של הפתרון). הפתרון הזה הוא יקר. אנו מעוניינים למצוא דרך מהירה יותר לעשות פעולה זו.

אינדקס Permuterm

נניח שיש לנו את ה-term hello. בכדי לבנות את ה-Permuterm אינדקס של ה-term הזה, נוסיף בסוף ה-term את הסימן המיוחד \$ (שמציין את סוף הביטוי), כלומר נקבל את ה-term hello\$. מה-term החדש אנו נבנה terms חדשים, כשכל term מבוסס על ה-term הקודם, והוא בנוי כך שלוקחים את התו הראשון של ה-term הקודם, ומעבירים אותו לסוף ה-term. כלומר, עבור ה-term hello, אנו מקבלים את ה-permuterms: hello\$, ello\$h, llo\$he, lo\$hel, o\$shell, \$hello

כל permuterm כזה נחשב כ-term עבור תהליך יצירת ה-inverted index (כשכולם למעשה מתייחסים לאותו term מקורי, ולכן מצביעים לאותו מסמך).

כאמור, המטרה שלנו היא לאפשר חיפוש עם wild-cards באופן מהיר. אנו עושים זאת באופן הבא:

- X\$, יש לבצע חיפוש עבור X\$
- *X\$, יש לבצע חיפוש עבור *X\$
- \$X\$, יש לבצע חיפוש עבור \$X\$
- *X\$, יש לבצע חיפוש עבור *X\$
- X*\$Y, יש לבצע חיפוש עבור X*\$Y

בכדי לבצע את השאילתא hel^*o , אנו רואים שהשאילתא היא למעשה בצורה X^*Y , ולכן $X=hel$, $Y=o$, ובהתאם נחפש במילון את ה-term (או ה-permuterm) ohel^*$.

שימו לב ששימוש ב-permuterm אינו דורש מאתנו להחזיק עץ "הפוך".

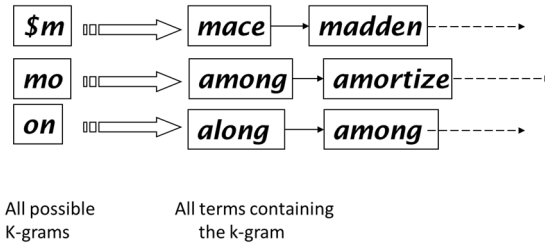
אינדקס (k-gram) Bigram

שיטה נוספת שמאפשרת לנו לבצע שאילתות עם wild-cards היא k-gram. k-gram הינו צרוף של k תווים במסמך שלנו, כאשר גם כאן אנו משתמשים בתו \$ בכדי לציין הפרדה בין terms שונים במסמל. בכדי לעבוד עם k-gram אנו צריכים, תחילה, למנות את כל ה-k-grams (צרופים של k אותיות) הקיימים בכל terms שלנו. לדוגמא, עבור הטקסט "April is the cruelest month", נקבל את ה-k-grams ($k=2$ bigram-ים) הבאים:

\$a, ap, pr, ri, il, l\$, \$i, is, s\$, \$t, th, he, e\$, \$c, cr, ru, ue, el, le, es, st, t\$, \$m, mo, on, nt, h\$

במקרה זה, אנו יוצרים inverter index מיוחד שהמילון שלו מורכב מ-k-grams, ואילו ה-posting list של כל ביטוי במילון, הוא רשימת terms המכילים את ה-k-gram הנ"ל.

הדוגמא הבאה מתארת אינדקס k-gram המתקבל עבור $k=2$ (bigram).



נניח ואנו רוצים לבצע את השאילתא *mon. את השאילתא הזאת אפשר לתרגם ל-\$m AND mo AND on. השאילתא הנ"ל תחזיר לנו את התוצאות העונות לשאלה *mon, אבל היא גם עלולה להחזיר תוצאות שאינן מתאימות לשאילתא המקורית שלנו, כמו moon. לכן, כל תשובה שמתקבלת, יש לבדוק התאמה מול השאילתא המקורית. את התוצאות שתואמות את השאילתא המקורית יש לחפש ב-inverted index של המושגים-מסמכים.

שיטה זו מהירה יותר וחסכונית יותר בזיכרון בהשוואה ל-permuterm.

בשאילתות כגון *pyth AND *prog, אנו נתייחס לכל שאילתת wild-card בנפרד. ועבור התוצאות שנקבל אנו נבצע שאילתת AND. ריצה של שאילתות כאלו, עלולה לקחת זמן רב.

Spelling correction

לתיקון שגיאות כתיבת שני שימושים עיקריים. (1) תיקון שגיאות כתיב במסמכים אותם אנו מאנדקסים. (2) תיקון שגיאות כתיב בשאילתות שהמשתמש מזין (על מנת לקבל תשובות נכונות).

תיקון שגיאות נעשה בשני אופנים עיקריים. (1) תיקון שגיאות של מילים בודדות - תיקון שגיאות כתיב של מילה בודדת. במקרים כאלו, לא נזהה שגיאות אם האיות של המילה תקין (from במקום form). (2) תיקון שגיאות בהתאם להקשר - כל מילה נבדקת ביחס למילים שנמצאות לידה. לדוגמא, I flew form Heathrow to Narita.

תיקון שגיאות נדרש במיוחד עבור מסמכים שנסרקו (OCR). במקרים כאלו, האלגוריתמים שמהים את האותיות, עלולים לטעות ולזהות אותיות (או צירופי אותיות) מסוימות כאותיות אחרות. לדוגמא, האותיות m ו-in דומות אחת לשנייה. במידה והסריקה לא באיכות טובה, או שהאלגוריתם אינו מדויק/רגיש מספיק, יהיו מצבים שהאות m תזהה כ-in ולהפך. אלגוריתמים לתיקון שגיאות בנויים לזהות שגיאות מסוג זה. הם כוללים "ידע מיוחד" הקשור לבעיות סריקה, כגון רשימה שלא אותיות הנראות דומה, כמו m ו-in או O ו-D (לעומת I ו-O, שהסיכוי לבלבול בניהם נמוך).

גם במסמכים מסוגים אחרים, למשל מוקלדים, יש שגיאות כתיב. למרות שהמטרה שלנו, שבמילון יהיו כמה שפחות שגיאות כתיב, בהרבה מקרים אנו לא "מתקנים" את המסמכים עצמם, אלה דואגים שהמיפוי בין השאילתא למסמך יהיה תקין.

נניח שאנו מחפשים את הביטוי Alanis Morisset. אנו יכולים (1) להחזיר את המסמכים המכילים את הכתיב הנכון של הביטוי, או (2) להחזיר מספר אפשרויות לשאילתות חדשות, המכילות תיקוני שגיאות כתיב (Did you mean ... ?)

כאשר אנו מבצעים תיקון של מילים בודדות, הנחת יסוד היא שיש לקסיקון שלפיו נקבע האיות הנכון של כל מילה. במקרה הזה, יש לנו שתי אפשרויות, (1) להשתמש בלקסיקון סטנדרטי, כדוגמת Webster's English Dictionary או An "industry-specific" lexicon - hand-maintained. (2) לבנות לקסיקון הבנוי במילים הנמצאות במסמכים המאונדקסים שלנו (הקורפוס). הלקסיקון הזה יכול את כל המילים ב-web ואת כל השמות (כולל מילים עם שגיאות כתיב).

בדיקת האיות נעשית באופן הבא. כאמור, נתון לנו לקסיקון וסידרה של תווים, Q (המילה אותה אנו רוצים לבדוק). המטרה שלנו היא למצוא את המילה בלקסיקון הקרובה ביותר ל-Q. אם קיימת מילה בלקסיקון, ש"המרחק" בינה לבין סידרת התווים שלנו, Q, הוא 0, הרי שהמילה שלנו מאויתת נכון. אחרת, ישנה שגיאת כתיב, וכנראה שהמילה הקרובה ביותר במילון היא הכתיב הנכון.

ובכן, כיצד אנו מגדירים את המושג "קרובה ביותר"? לשם כך נדבר על מספר אלטרנטיבות: (1) Edit distance (Levenshtein distance), (2) Weighted edit distance ו-n-gram overlap (3).

Edit distance

נתונות שתי מחרוזות, S_1 ו- S_2 , edit distance מוגדר כמספר הפעולות (דרמת התווים – הוספה, מחיקה ושינוי תווים) המינימאלי הנדרש על מנת להגיע מהמחרוזת הראשונה לשנייה.

לדוגמא:

- מ-dof ל-dog, זה 1
- מ-cat ל-act, זה 2
- מ-cat ל-dog, זה 3

החישוב של edit distance לרוב ממומש ע"י תכנות דינאמי (<http://www.merriampark.com/ld.htm>).

Weighted edit distance

בדומה ל-Edit distance, weighted edit distance מוגדר כמספר הפעולות המינימאלי הנדרש על מנת להגיע מהמחרוזת הראשונה לשנייה, אולם לכל פעולה יש משקל התלוי בתווים המשתתפים באותה פעולה. הרעיון הוא, שבכל סוג של מסמך יש טעויות שהן נפוצות יותר ונפוצות פחות. במסמכים סרוקים, הסיכוי להחליף בין m ל-rm גבוה יותר (משקל נמוך יותר) מאשר הסיכוי להחליף בין m ל-q (משקל גבוה יותר). באופן דומה, במסמכים מודפסים יש סיכוי גבוה יותר לשגיאות הנובעות מטעויות הקלדה (החלפת סדר אותיות, או הקלדה על אות "קרובה" במקלדת). בכדי לחשב את ה-weighted edit distance אנו צריכים שתהיה לנו מטריצת משקלים, שמציינת את המשקל של החלפת אות אחת באות שנייה בהתאם להסתברות לבצע טעות כזו.

החישוב של ה-weighted edit distance דורש שינוי (קל) באלגוריתם התכנות הדינאמי שלנו.

השימוש ב-edit distance או weighted edit distance מתבצע באופן הבא. בהינתן שאילתא, תחילה אנו מונים את כל צירופי התווים שיש להם (Weighted) Edit distance נתון (למשל 2). נבצע הצלבה בין התוצאות שקיבלנו לרשימה של מילים "נכונות", בהתאם יש לנו שלוש אפשרויות. (1) נציג למשתמש את המילים שנמצאו (הצעות). (2) נבצע שאילתא המכילה את כל המילים שמצאנו (איטי). (3) נשתמש באפשרות הסבירה ביותר במקום כל המילים (אלו שהשיחות שלהם היא הגבוהה ביותר). בצורה זו אנו "חוסכים" את הצורך במשתמש (לא תמיד לטובה), ובכך מספקים תשובות מהר יותר.

כאמור, בהתחלה אנו מונים את כל צירופי התווים שיש להם (Weighted) Edit distance נתון. למעשה אנו יוצרים הרבה מאד צרופים לכל term בשאילתא. אפשרות אחרת, במקום ליצור צרופים חדשים, נחשב את ה-edit distances בין כל term בשאילתא לכל term במילון. התהליך הזה הוא מאד איטי, ולכן האם אנו באמת צריכים לחשב את ה-edit distances לכל מושג במילון ?

n-gram overlap

אנו יכולים להקטין את קבוצת ה-terms של המילון המעומדים לבדיקה (כלומר מולם נחשב את ה-edit distance), למשל ע"י שימוש ב-n-grams overlap. שיטה זו מאפשרת בדיקת איות גם עבור השאילתות עצמן.

שיטה זו פועלת באופן הבא. תחילה אנו מונים ה-n-grams במחרוזת השאילתא וכן בלקסיקון. אחר-כך אנו משתמשים באינדקס ה-n-grams (חיפוש עם wild-cards) בכדי להביא מהלקסיקון את כל המושגים התואמים ל-n-grams שבמחרוזת השאילתא. אנו קובעים סף המבוסס על מספר ה-n-grams התואמים (אפשרויות נוספות, להשתמש במשקלים המותאמים לשגיאות הקלדה וכו'), שלפיו אנו מחליטים אם הכתיב של term הוא תקין או לא, ובמידה ולא, איזה terms מהמילון יכולים להוות אלטרנטיבות לתיקון הכתיב השגוי.

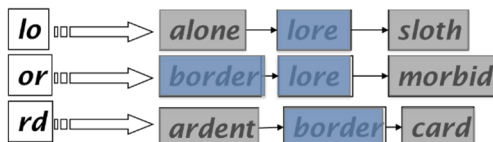
לדוגמא, נניח שהטקסט שלנו הוא November. בהתאם, אנו יכול לחשב את ה-3-grams הבאים: nov, ove, vem, emb, mbe, ber. נניח שהשאילתא שלנו היא December. עבורה ה-3-grams הם: dec, ece, cem, emb, mbe, ber. אנו רואים שיש התאמה של 3-grams. המספר הזה אינו עוזר לנו כי הוא יכול להיות שונה בין terms שונים, ואז תהייה לו משמעות שונה. צריכים להפוך את הערך הזה לערך מנורמל.

שיטה אחת לנירמול היא Jaccard coefficient. נניח ש-X ו-Y הן שתי קבוצות, J.C. מוגדר כ:

$$\frac{|X \cap Y|}{|X \cup Y|}$$

באופן זה, אנו מקבלים 1 כאשר X ו-Y (אשר אינם חייבים להיות באותו הגודל) מכילים את אותם הערכים, ו-0 כאשר הם שונים לגמרי. הערך המתקבל הוא תמיד בין 0 (כולל) ל-1 (כולל). באופן כזה אנו יכולים לקבוע סף, למשל, אם J.C גדול מ-0.8, אנו מתייחסים לזה כהתאמה.

נניח שהשאלתא שלנו היא lord, ואנו מעוניינים במילים שיש בהם לפחות 2 מ-3 ה-bigrams שבביטוי (lo, or, rd). אנו נשתמש בערך מנורמל (J.C. או אחר) בכדי לקבוע התאמה.



תיקון שגיאות בהתאם להקשר

נתון הטקסט: I flew from Heathrow to Narita. וכן, נתונה השאלתא: flew form Heathrow. אנו רוצים לשאול את המשתמש אם הוא התכוון ל-flew from Heathrow, מכיון שאין מסמכים המכילים את השאלתא. מכיון שאין שגיאות במילים עצמן, ניתן להבין שהמילה form שגוייה רק לפי ההקשר (המילים שלידה).

כדי לעשות זאת נפעל לפי השלבים הבאים. (1) לכל term בשאלתא נביא את ה-term-ים הקרובים לו (למשל ב-weighted edit distance). (2) נבדוק את התוצאות המתקבלות עבור כל השאלתות, כאשר כל שאלתא היא אחת מהפרמוטציות האפשריות.

אם נניח שבשאלתא יש רק מילה אחת שגוייה, מספר השאלתות הנבדקות יהיה קטן יותר. במקרה זה נבדוק את התוצאות המתקבלות עבור כל השאלתות, כאשר בכל שאלתא אנו מחליפים term אחד ב-term "מתוקן". למשל, (1) flew from Heathrow, (2) fled form Heathrow, (3) flea form Heathrow וכו'. אנו נציע למשתמש את השאלתא שמחזירה את מירב התוצאות.

כאמור, יש לנו מספר אפשרויות ל"האם התכוונת ל...?", ואנו צריכים החליט אלו מהאפשרויות להציג למשתמש. בדוגמא הקודמת אמרנו שניציג למשתמש את השאלתא מחזירה את מירב התוצאות, אולם לפעמים אנו יכולים לבצע ניתוח של שאלתות עבר ולהחליט על סמך זה. באופן כללי יותר, אנו רוצים לדרג את ההסתברות לקבלת תשובה נכונה (כלומר, $argmax_{corr} P(Corr|Query)$ או בהתאם לחוק Bayes, $argmax_{corr} P(Query|Corr) \times P(Corr)$ - אנו נדבר על נושאים אלו בהרצאה אחרת).

Soundex

סאונדקס היא היריסטיקה שמטרתה להפוך מילה או שאלתא למקבילה הפונטית שלה. כל token במסמך מועבר למצב "מוקטן" הבנוי מ-4 תווים, ומאונדקס לפיו. את אותה הפעולה אנו עושים לכל term בשאלתא. החיפוש נעשה בהתאם לצורת ה"מוקטנות" כאשר החיפוש הוא לפי סאונדקס.

אלגוריתם סאונדקס כללי

1. Retain the first letter of the word.
2. Change all occurrences of the following letters to '0' (zero):
'A', 'E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows:
 - B, F, P, V → 1
 - C, G, J, K, Q, S, X, Z → 2

- D,T → 3
 - L → 4
 - M, N → 5
 - R → 6
4. Remove all pairs of consecutive digits.
 5. Remove all zeros from the resulting string.
 6. Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter> <digit> <digit> <digit>.

E.g., **Herman** becomes H655.

לדוגמא, נסתכל על המילה Herman. בשלב הראשון אנו שומרים את האות הראשונה של המילה, כלומר, Herman. בשלב השני, כל המופעים של האותיות e ו-a מוחלפים ב-0, בהתאם לסעיף 2 של האלגוריתם, ואנו מקבלים H0rm0n. בשלב השלישי אנו מחליפים את האות r ל-6, ואת האותיות m ו-n ל-5, ומקבלים H0650n. לאחר מכן אנו צריכים להסיר תווים זהים רציפים, אולם במקרה שלנו אין לנו. בשלב הבא אנו מורידים את ה-0ים שהופיעו, ומקבלים H655. בשלב האחרון אנו לוקחים רק את 4 האברים השמאליות ביותר. כלומר, התוצאה הסופית היא H655.

סאונדקס הוא אלגוריתם כללי, שקיים כמעט בכל בסיס נתונים (מיקרוסופט, אורקל...). עד כמה הוא יעיל? לא הרבה בתחום של אחזור מידע. כן ניתן למצוא לו שימוש במקרים מיוחדים (למשל שמות). קיימים אלגוריתמים אחרים בתחום הפונטיקה שהם יותר ישימים בתחום של אחזור מידע (Zobel and Dart, 1996).

תרגיל: יש להפעיל את האלגוריתם על שתי המילים הבאות: Peking ו-Beijing.