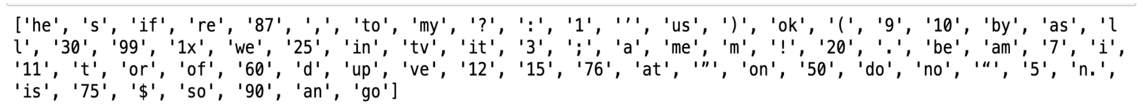
Ashkan Zarkhah
610399196

- **Documentation**

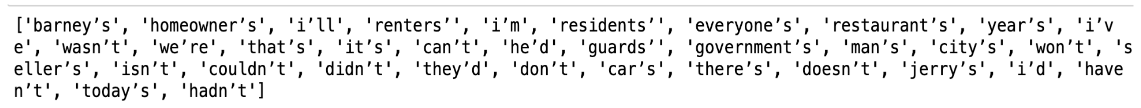
  a. **Preprocessing**

  First, we loaded our data from Disk to our memory. All documents are stored in the docs folder so we loaded each one of them from the “docs” folder and we stored them in the “documents” variable.
  Then we did our first preprocessing and we lowercased all of our documents so that words like “I” and “i” didn’t differ and we stored the lowercase documents back in the “documents” variable.
  Then we tokenized our documents with the help of the “nltk” library and saved each tokenized document independently in the “tokenized_documents” variable.
  Then we printed small tokens out to see if we had any non-meaning tokens. After checking the tokens we found a lot of problems. There were so many tokens that didn’t have any meaning like “ve” or “m”.



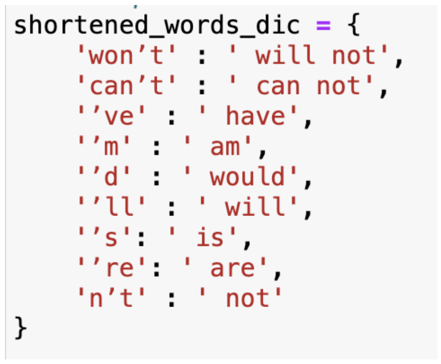
```
['he', 's', 'if', 're', '87', ',', 'to', 'my', '?', ':', '1', '’', 'us', ')', 'ok', '(', '9', '10', 'by', 'as', 'l
l', '30', '99', '1x', 'we', '25', 'in', 'tv', 'it', '3', ';', 'a', 'me', 'm', '!', '20', '.', 'be', 'am', '7', 'i',
'11', 't', 'or', 'of', '60', 'd', 'up', 've', '12', '15', '76', 'at', '”', 'on', '50', 'do', 'no', '“', '5', 'n.',
'is', '75', '$', 'so', '90', 'an', 'go']
```

  At first sight, all non-meaning tokens were small, so we listed and uniqued them to see what they were. After analyzing them a little, we found out that they are the shortened term, and in our tokenization, they were split because of “ ’ ”.
  Then we used regex to find all split by “ ’ ” words and listed and uniqued them to see all needed shortened forms.



```
['barney’s', 'homeowner’s', 'i’ll', 'renters’', 'i’m', 'residents’', 'everyone’s', 'restaurant’s', 'year’s', 'i’v
e', 'wasn’t', 'we’re', 'that’s', 'it’s', 'can’t', 'he’d', 'guards’', 'government’s', 'man’s', 'city’s', 'won’t', 's
eller’s', 'isn’t', 'couldn’t', 'didn’t', 'they’d', 'don’t', 'car’s', 'there’s', 'doesn’t', 'jerry’s', 'i’d', 'have
n’t', 'today’s', 'hadn’t']
```

  After finding all the shortened forms, we decided that the best way to solve this problem was to convert all of our shortened forms back into their original form. So we made a dictionary from every shortened form to their original form and we put special cases first so that if it’s one of them it changes first and then we replaced every one of them with the original form and stored them back in the “tokenized_documents” variable.



```
shortened_words_dic = {
    'won’t' : ' will not',
    'can’t' : ' can not',
    '’ve' : ' have',
    '’m' : ' am',
    '’d' : ' would',
    '’ll' : ' will',
    '’s': ' is',
    '’re': ' are',
    'n’t' : ' not'
}
```

After the shortened forms, we tried again to find the short non-meaning tokens and saw that they were punctuation marks.

```
['he', 'if', '87', ',', 'to', 'my', '?', ':', '1', 'us', ')', '’', 'ok', '(', '9', '10', 'by', 'as', '30', '99', '1
x', 'we', '25', 'in', 'tv', 'it', '3', ';', 'a', 'me', '!', '20', '.', 'be', 'am', '7', 'i', '11', '60', 'or', 'o
f', 'up', '12', '15', '76', 'at', '”', 'on', '50', 'do', 'no', '“', '5', 'n.', 'is', '75', '$', 'so', '90', 'an',
'go']
```

So we filtered them out of our tokens too and stored filtered tokens back in the “tokenized_documents” variable.
After that, we checked if there was any punctuation mark or dashes left in our tokens. We saw that there were plenty of dots and question marks and all kinds of dashes left at the end or in the middle of the tokens. This was because words with dashes were not split by NLTK tokenizer and the last token of each sentence was with its punctuation mark and for sentences that didn’t start with a blank space, the last token of the last sentence and the first token of the current sentence wasn’t split.

```
['available.this', 'question–and', 'drinks.people', 'too.', 'murder–suicide', '74–year', '70–year', 'next–door', 'm
rs.', 'mr.', 'gone.', 'mrs.', 'quieter.', 'pain.', 'night–', 'money.sam', 'three–hour', 'decision.finally', 'sixty–
year', 'car.sam', 'inc.', 'donor.', 'inc.', 'inc.', 'inc.', '000.', 'year.', 'california.', 'two–bedroom', 'one–bed
room', 'looking.', 'opportunity.', 'paper.the', '24–year', 'one–hour', 'vehicle.when', 'halt.turning', 'chase.the
y', '2.22', '2.09', 'customers.the', '2.14', 'money.', 'door–slammer', 'job–related', 'middle–aged', 'tune–ups', 'b
inoculars.', 'bus.jerry', 'guy.jerry', 'knee.jerry', '79–year', 'drive–through', '9:00', 'p.m', 'scalding.he', 'ext
ra–large', 'mcrap.the', 'time.', 'clubs.much', 'groups—the', 'groups—save', 'two–mile', 'filled.no', 'five–year',
'triple–scoop', '15–percent', 'more.', 'two–', 'three–bedroom', 'sidewalks.noise', 'ban.city', 'them.some', 'n.',
'1992.', 'four–door', 'a.m', 'away.barget', 'four–slice', '29.95', '39.95', '3.50', 'change.sara', 'a.m', '000.resi
dents', 'playground.', 'operation.ninety', 'necessary.', '65–year']
```

Then we split every token with a dot (we didn’t split numbers) or question mark or any kind of dash in the middle of it and also removed dots and question marks and all kinds of dashes from the end of the tokens.
Now it was time for our next preprocessing which was finding and removing stop words. Normally stop words are the most frequent words so we gathered all the tokens in the “all_tokens” variable and then we sorted it so that we could count how many of each token we had. After counting and storing each token with its frequency in the “counted_tokens” variable, we sorted this one too and we found our 10 most frequent tokens.

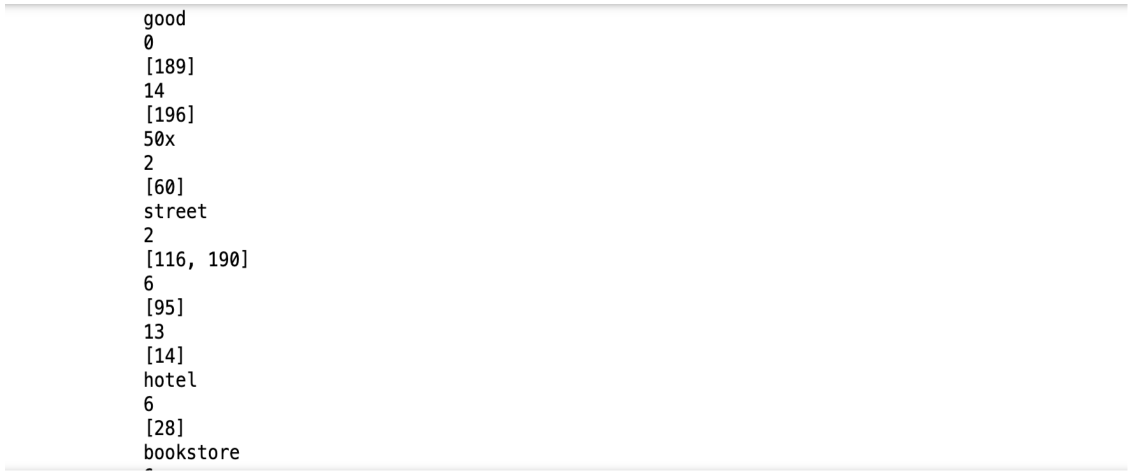
```
[[46, 'he'], [48, 'for'], [60, 'in'], [64, 'is'], [68, 'was'], [77, 'of'], [85, 'and'], [114, 'to'], [141, 'a'], [2
21, 'the']]
```

As we expected they were all stop words so we filtered them all out of our “tokenized_documents” variable (but kept a copy for our misspelling check) and stored the filtered one back in its place.
After preprocessing tokens, we checked them one last time and they were all good.

b. Inverted Index

After preprocessing the tokens, we needed a hash table for our inverted index and for the hash table, we needed to know how many different tokens we had. So we gathered all our tokens and uniqued them in the “all_tokens” variable. After finding that we have a total of 1292 tokens, we developed our hash function which would hash every token into an independent number from 0 to 1291. Our hash function first calculates the token’s number in 256 base and module it by 1292 so that its number is in the wanted range and if that cell is full, it will loop

from that point till it finds an empty cell and it will store the token in that cell and from now that cell’s index is our hash for the token.
After having the hash function it was time for building a structure for our posting list. We decided that a linked list is the best data structure for it. So first we defined a node for each token happening in each document, or in other words for each index(happening) in a document and then we defined a document_node which was a linked list of all happenings of a token in a document and we defined two functions for it, first one inserts a happening at the end of the liked list and second one returns a list which contains all of the happenings in it. After that, we define a posting list which is a linked list of document nodes and each token would have a posting list for itself and it contains each document of the token’s happenings and within each document_node it has all of its happenings in that document. We defined two functions for posting lists too, which were similar to the functions of document_node. The first one inserts an index in document_node of the list and the second one gives us all documents of a token.
After developing the posting list structure we made 1292 individual posting lists, one for each token, and then it was time to go through every document and each token of it and add the index of that token’s happening in the document_node of its posting list.
After adding document IDs to the posting list, we printed them out to make sure they were filled correctly.



```
good
0
[189]
14
[196]
50x
2
[60]
street
2
[116, 190]
6
[95]
13
[14]
hotel
6
[28]
bookstore
```

c. **Spelling Correction**
Now it was time to define a function that would find all misspelling candidates for us. As it was said in the homework description we decided to use Levenshtein distance. So first of all we needed to define a function that would get two words and return their distance. So we defined the “calculate_dis” function and within it, we defined a 2D array “dis” that would have distance values in it. We put initial values equal to the sum of input lengths because the distance of two words is at most sum of their lengths. After that, we filled our “dis” array from smallest “i” to largest “i” and for each “i” from smallest “j” to largest “j” and we minimized its value with any possible recurse for it. After defining the distance function, it was time to find all the closest tokens to any input word. So we searched all tokens

with the same starting character(our heuristic) and calculated their distance and saved all minimum distances in the “selected_tokens” variable and then returned it.

d. **Wildcard Queries**

Now it was time to define a function that would find all wildcard candidates for us. But first, we needed another linked list because we wanted to use the 2-gram algorithm. Our new liked list would contain every token of every one of our 2-grams. So first we defined a node for it which contains its token and a pointer to its next node and then we defined a linked list that would contain all tokens of a 2-gram and we defined two functions for it. The first one inserts a new token at the end of a 2-gram’s list and the second one returns all of a 2-gram’s tokens. After building the KGList, we defined a 256 * 256 array of KGLists to have a KGList for every possible 2-grams. After that, we looped over our tokens and added a ‘#’ before and after it (to separate the token's begin and end) and added this token to every 2-grams of it.

Now that we had our KGLists filled, it was time to define a function that would find all wildcard candidates of a given word. To find them first we assumed that all tokens were candidates and then for every 2-grams of the input we got its tokens and we replaced our candidates with their intersection. After getting all 2-gram’s candidates and getting their intersection it was time to remove false positives, so we checked if the token prefixes and suffixes were the same as the inputs, and if the input had 2 stars we checked if the middle part was present in the tokens too. And after filtering all false positives we return our candidates. Note that we kept our candidates sorted to have an optimized intersection function.

e. **Query Processing**

Now that everything was ready it was time to handle queries. We had 4 types of queries and we defined four individual functions to handle them.

1. And_handler: In this function, we got document lists of both inputs and if any of them were not one of our tokens, the answer was an empty list, so we checked that first. After making sure both of the documents were not empty we used a two-pointer algorithm to find their intersection.
2. Or_handler: In this function, we got document lists of both of inputs and if any of them were not one of our tokens we put an empty list instead of it. After making sure both of the documents were defined we added the first one at the end of the other one and uniqued it.
3. Not_handler: In this function, we got the document list of input and if it was not one of our tokens, we put an empty list instead of it. Then we used a two-pointer algorithm to find indexes that were not in the document list and we returned them all.
4. Near_handler: In this function, we got document lists of both inputs and if any of them were not one of our tokens, the answer was an empty list, so we checked that first. After making sure both of the document lists were not empty we used a two-pointer algorithm to find their intersection. After finding each intersection it was time to check if there was any happening

    with at most k distance in both token's happenings. To check that we used a two-pointer algorithm to find each closest bigger happening of the second token's happenings in the first token's happenings and we checked if this happening of the first token or the one before it has the needed distance or not. And after finding one, we added the document's id and moved on to the next intersection.

After defining handlers, we defined a general function to decide the query's type. In this function we first tokenized our query and then lowercased our inputs and then for each token we had 3 possible values:

1. It was AND, OR, NOT, or NEAR/K, in that case, we would just append a one-member list to our all possible queries list.
2. It was a wildcard, in that case, we would append a list of all candidates instead of the wildcard in our all possible queries list.
3. Otherwise, we would check if it has a misspelling and we would append a list of all misspelled candidates instead of the word in our all possible queries list

Then when we had our all possible queries list, we would loop over it and print each possible query and its output from its query time's handler, and after showing all possible queries and their answers, we would answer union of them too.
And then we ran a true loop to answer any query.
Also, we have added two additional blocks at the end of the code to check misspellings and wildcards individually.

- **Report:**
  Because we have described our code and our problems and their solutions completely in the documentation part, we just summarize them here.
  - **Key findings:**
    - NLTK tokenizer can't handle shortened forms in the English language and also it assumes that sentences always start with a blank space in the first and it leaves each sentence's ending punctuation mark with the token and it doesn't tokenize dashes..
    - We can't generalize all shortened forms and there are a few exceptions, like "won't".
    - For misspelling check, we must keep all our tokens and don't filter our stop words.
    - Not all heuristics for misspelling are faster than calculating their distance. For example, our first heuristic was if they differ more than 4 characters we filter those tokens and don't calculate distance for them but calculating the heuristic was slower than calculating distance itself.
    - Because we have too many different 2-grams, it is better to keep a 2D array for each KGList than to have a linked list for them together.
    - We must apply every preprocessing we have done with our tokens, on our queries too.
  - **Challenges faced:**

    - Checking if the tokens were correct was a time-consuming job and there weren't any algorithmic ways for it.
    - Finding the best way to generalize shortened forms was also troublesome.
    - Finding a clean way to check false positives of the 2-gram algorithm was hard and troublesome.
    - Developing a two-pointer algorithm for the Near_handler function was hard and debugging it was hard too.
    - Generating all possible queries was very specific on a query and it was hard to generalize it,
  - **Enhancements:**
    - We developed a fast heuristic for our misspelling algorithm so it doesn't calculate distance for every token.
    - We kept our candidate_tokens sorted in our wildcard finding and it let us use the two-pointer algorithm for our intersection finding and decreasing our time complexity from O(NM) to O(n + m). (N and M are the length of input arrays)
    - We used a 2D array to find our KGLists and because of the array's random access, finding each KGlist's time complexity decreased from O(N) to O(1). (N is the number of 2-grams)
    - For AND and NEAR handlers we developed a two-pointer algorithm instead of finding intersections through brute force and it helped us to decrease our algorithm time complexity from O(NM) to O(N + M). (N is the number of happenings of the first token and M is for the second one)

- **References**
  - **os library (for including documents)**
  - **NLTK library (for tokenizing)**
  - **re library (for finding shortened forms)**