
How Research Works

An "Under the Hood" Exploration of Search Algorithms

Vagelos Computational Science Center - Barnard College
March 12, 2025
Milstein 516

Introductions

Dan Woulfin

Computational Research Instruction Librarian
Columbia University

Sydni Meyer

Teaching and Undergraduate Services Librarian
Columbia University

The experiment for today

Can we work at the
**intersection of
computational and
information literacy** in
order to shine a light on
the mechanics behind
academic database and
text based search?

Agenda

*Please feel free to interrupt us at any time.
We want this to be conversational and
welcome your thoughts, questions, and
intrusions.*

1. Introduction
 2. Library Database Search
 3. The Mechanics of Boolean Search
 4. Measuring relevance and ranking results
-

Information and Computational Literacy

	Vertical Reading	Lateral Reading	Proactive Reading
Information	Looking closely at one source and evaluating it as a single entity	Evaluating a source with other/multiple sources	Asking why something is included and what might be missing or hidden.
Computation	Looking only at the code	Digging into the functions, libraries, and documentation of the code to figure out how it works	Asking why the programmer made their decisions and if they were the right ones.
TLDR	Surface level analysis	Comparative analysis	Analyzing Intent

Bull, A. C., MacMillan, M., & Head, A. (2021). Dismantling the evaluation framework. *the Library with the Lead Pipe*.
<https://www.inthelibrarywiththeleadpipe.org/2021/dismantling-evaluation/>

How computers “read”

A computer only “sees” or “reads” sequences of characters. It needs to be told how to parse it:

- Tokenization
- Part of Speech tagging
- Processing: stopword removal, lemmatization, stemming

It doesn't know words, sentences, idioms, metaphors, paragraphs, ideas, phrases, etc. To a computer it is just a sequence of characters.

This is not a sentence.

This is not a “word”.

This is not an idea.



Library Database Search

What is with this boolean thing?

From human to menu to algorithmic interfaces

1970s	1980s	1990s-2000s
Access to online databases were very expensive and are stored in the library or other institution	Proliferation of PCs and storage means that databases (authorities) can be stored and accessed at home	The birth of the internet means that information is held offsite and accessed via search engines
Librarians served as the human interface between the databases (authorities) and the user	Creation of a menu interface , limiting human connection and search options	Creation of recommendation algorithms . Authority is defined by the number of links to a site.

Kerssens, N. (2017). When search engines stopped being human: menu interfaces and the rise of the ideological nature of algorithmic search. *Internet Histories*, 1(3), 219-237. <https://doi.org/10.1080/24701475.2017.1337666>

The Librarian Database Demo

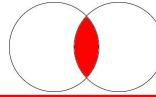
Search operators

Filters and fields

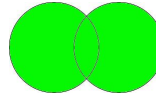
Controlled vocabularies

Search operators with truth tables

AND: Returns the intersection of two sets or all the elements that appear in both sets.



OR: Returns the union of two sets or all elements that appear in either or both sets.



NOT: Returns the negation of a set or anything not in the set.



\mathcal{A}	\mathcal{B}	$\mathcal{A} \& \mathcal{B}$	$\mathcal{A} \vee \mathcal{B}$
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

\mathcal{A}	$\neg \mathcal{A}$
1	0
0	1


Academic database structure

Every database is structured differently.

For example:

- Clarivate's Web of Science, Elsevier's Scopus, Ithaka's JSTOR, and Digital Science's dimensions.ai have different field names, datasets, and rules
 - We don't know how they store their data and whether it's a relational database, document store, graph network or something else
-

Break down what we just saw using computational thinking

	Decomposition: Break down a problem into smaller problems or tasks	Process individual queries into AND, OR, NOT, words/phrases, parentheses, *, ? and run it against document(s) producing results.
	Pattern recognition: Identify patterns or similarities within the broken down tasks/problems	Queries use boolean operators, wildcards, & words/phrases. They are applied to a document using an order of operations.
	Abstraction: Find the essential characteristics in the patterns to find core concepts.	Methods like regular expressions can handle matching, while conditionals can handle boolean operations / the order of operations.
	Algorithmic design: Design step-by-step procedures (AKA algorithms) from the abstractions.	See next slide
	Review and re-evaluate: Regularly streamline and finetune these solutions	Future?

The algorithm

Read in the documents and
query

Tokenize the query and add
implicit booleans

Process the query against
each document

Return matches/positive
results

The mechanics of boolean search

Tracing the code

Follow along at <https://bit.ly/csc-search-wkshp-2025-03>

Understanding boolean search

We will now go through a boolean search algorithm using Python function by function. The functions have been prewritten for for three reasons:

Practice **tracing** the flow of the program, understanding the logic without running the code (**unplugged learning**)

Allow for questions, discussion, and interruptions rather than spending time debugging

Avoid Dan's many many many typos and tangents while live coding

Why functional programming?

I've set up the program design as functions for three reasons:

Functional programming
is based on
reproducibility. A
function always provides
the same result given the
same arguments.

Reusable functions might
come in handy.

Object Oriented
Programming gives me a
headache.

This is probably the wrong technical decision for Python.

Pattern matching

There are multiple ways to match a query to a document including:

- Exact matches
- Regular expressions
- Fuzzy matching (approximate/phonetic matching)

We'll be using regular expressions because they seem to be what's used in academic databases.

Regular expressions (regex)

These are formulas that allow you to find matches in a string or document.

Groups and Ranges

.	Any character except new line (\n)
(a b)	a or b
(...)	Group
(?:...)	Passive (non-capturing) group
[abc]	Range (a or b or c)
[^abc]	Not (a or b or c)
[a-q]	Lower case letter from a to q
[A-Q]	Upper case letter from A to Q
[0-7]	Digit from 0 to 7
\x	Group/subpattern number "x"

Ranges are inclusive.

Quantifiers

*	0 or more	{3}	Exactly 3
+	1 or more	{3,}	3 or more
?	0 or 1	{3,5}	3, 4 or 5

Add a ? to a quantifier to make it ungreedy.

No one remembers regex.
We all look it up and
experiment on something
like regex101.com.

Tokenizing the query using regular expressions

Tokenization is the process of splitting characters. There's a lot of ways to tokenize text.

What regex patterns am I looking for and why? What am I ignoring?

How am I preprocessing the text with string methods?

```
import re

# Tokenize the query into words, phrases, and operators
def tokenize_query(query):
    # Define regex patterns for various token types
    token_patterns = [
        r'\(', # Parenthesis
        r'\)', # Parenthesis
        r'"[^"]+"', # Quoted strings
        r'[^) ]+', # everything that's not a space or a right parenthesis, including operators and wildcards
        r'\s+', # Whitespace (to be ignored)
    ]

    # Combine the patterns into one single pattern
    combined_pattern = '|'.join(token_patterns)

    # Use finditer for better control (positioning and matching)
    tokens = []
    for match in re.finditer(combined_pattern, query):
        token = match.group(0).strip().lower() # Get the matched text and normalize it
        if token.strip(): # Ignore empty tokens
            tokens.append(token)
    return tokens
```

Cleaning/Transforming the Query

Cleaning text is a crucial preprocessing step when working with text.

What do we do when people just type in words without a boolean operator?

We have to add OR in between.

Similarly NOT has an implicit AND before it.

```
# Handle implicit OR (e.g., "word1 word2" should be interpreted as "word1 OR word2")
# Also implicit AND before NOT unless the query starts with NOT
def add_implicit_boolean(tokens):
    processed_tokens = []
    prev_was_term = False # Tracks if the last token was a word/phrase

    for token in tokens:
        if token not in {"and", "or", "not"} and token not in {"(", ")"}:
            if prev_was_term:
                processed_tokens.append("or") # Insert implicit OR
                prev_was_term = True # Mark that the last token was a term (word/phrase)
            elif token == "not":
                # If the previous token is a term, insert implicit AND before NOT
                if prev_was_term:
                    processed_tokens.append("and") # Add implicit AND before NOT
                    prev_was_term = False # Reset, as NOT is an operator
            else:
                prev_was_term = False # Reset if an operator or parenthesis is found

        processed_tokens.append(token)

    return processed_tokens # Return the processed tokens
```

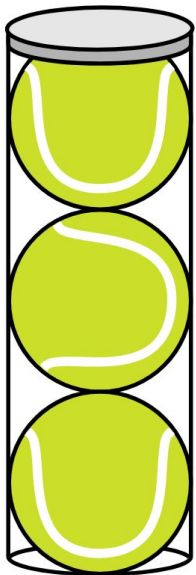
Setting up: Processing the query against a document

We need a way to process the cleaned query against the document. To do so we'll need

- 1) A way to evaluate tokens as booleans or operations
- 2) Maintain the order of operators and the order of tokens (stacks)

```
def process_query(query, document, partial = False):  
    tokens = tokenize_query(query)  
    tokens = add_implicit_boolean(tokens)  
  
    bool_results_stack = [] # Stores boolean results  
    operator_stack = [] # Stores operators
```

Last in, First Out



What's a stack

Query: not(apple and banana)

Document: I like apples and oranges.

Build the stack

token	not	(apple	and	banana)
results_stack	[]	[]	[True]	[True]	[True, False]	[True, False]
operator_stack	[not]	[not, (]	[not, (]	[not, (, and]	[not, (, and]	[not, (, and,)]

Evaluate the stack

Evaluation: Start at the end. Evaluate everything from the closed to the empty parenthesis (True and False). Then negate with not.

Building the stacks

What happens with:

- 1) Open Parenthesis
- 2) Closed Parenthesis
- 3) “not”
- 4) “and” & “or”
- 5) Phrases in quotes
- 6) Unigrams (single words)

Question: What's missing in the code? What have we not seen yet?

```
for token in tokens:
    if token == "(":
        operator_stack.append(token)
    elif token == ")":
        # evaluate everything in parenthesis
        while operator_stack and operator_stack[-1] != "(":
            operator_stack, bool_results_stack = apply_operator(operator_stack, bool_results_stack)
        if operator_stack:
            operator_stack.pop() # Remove "("
    elif token == "not":
        operator_stack.append(token)
    elif token in {"and", "or"}:
        # apply not first
        while operator_stack and operator_stack[-1] == "not":
            operator_stack, bool_results_stack = apply_operator(operator_stack, bool_results_stack)
        # apply AND/OR next from left to right
        while operator_stack and operator_stack[-1] in {"and", "or"}:
            operator_stack, bool_results_stack = apply_operator(operator_stack, bool_results_stack)
        operator_stack.append(token)
    else:
        if token.startswith("'") and token.endswith("'"):
            phrase = token.strip("'")
            bool_results_stack.append(phrase in document)
        else:
            document_words = document.split()
            bool_results_stack.append(word_matches(token, document_words, partial))
```

How to handle wildcards, truncation, and punctuation

regex comes to the rescue. We can transform the search symbols into regex so that they recognize patterns rather than exact sequences.

```
# Returns full and partial matches of a token from document_words
def word_matches(token, document_words, partial = False):
    if "*" in token or "?" in token: # Handle wildcards
        regex = token.replace("*", ".*").replace("?", ".") # Convert to regex
        return any(re.match(regex, word) for word in document_words)
    if partial: # if user chooses partial matches
        # Escape only the period character
        if "." in token:
            token = token.replace(".", r"\.") # Escape the period by adding a backslash
        # returns True for partial matches
        return any(re.match(token, word) for word in document_words)
    else: # if user chooses full matches
        # remove punctuation to ensure exact matches
        document_words = {re.sub(r"[^\w\s]", "", word) for word in document_words}
        return token in document_words # Return True for exact matches
```

Questions: When would we use wildcards? Why am I giving two options in terms of matches? What aspects of a library database am I missing?

Controlled vocabularies

Predefined groups of categories usually in a hierarchy by librarians or other experts that are related to one another.

Broader than	Belonging to a larger group
Narrower than	A subset of the group
Equivalent To (Use for)	Synonyms
Related To	Lateral relationship but not necessarily direct relationship

How can these affect the search and how could they be coded in at this stage?

apply_operator(): Setting the boolean order of operations

The if-else statement, the conditional, ensures that NOT is always evaluated before AND and OR.

Question: Why do I not need the return statement for this to work? What makes lists special in Python?

```
def apply_operator(operator_stack, bool_results_stack):
    if not operator_stack:
        return bool_results_stack # No change if empty

    operator = operator_stack.pop()

    if operator == "not":
        result = evaluate(operator, bool_results_stack.pop())

    else:
        operand2 = bool_results_stack.pop()
        operand1 = bool_results_stack.pop()
        result = evaluate(operator, operand1, operand2)

    bool_results_stack.append(result) # Push result back onto stack
    return operator_stack, bool_results_stack # Return updated stacks
```


Evaluating boolean operations with Python commands

Conditionals will allow us to evaluate the boolean operators easily.

- Check for “not” and a single operand (boolean value).
- Check for “and” and two operands (boolean values) that surround it
- Check for “or” and two operands (boolean values) that surround it

Returns the result of the boolean operation.

```
# Evaluate a Boolean operation
def evaluate(operator, operand1=None, operand2=None):
    if operator == "not":
        return not operand1
    if operator == "and":
        return operand1 and operand2
    if operator == "or":
        return operand1 or operand2
```

Question: Are these the only operations in a library database? Has anyone ever used something else?

Finishing up boolean search

All we need is one more function to make the actual script super clean and easy to use. The function just searches the documents.

It loops through the documents, adds the True results to a results list, and returns the positive results.

```
# Evaluate the query across all documents
def search_documents(query, documents):
    results = []
    for doc in documents:
        if process_query(query, doc.lower()):
            results.append(doc)
    return results
```

Let's demo all this in the Jupyter notebook and talk about what decisions were made, why they were made, and what can't boolean search do?

Relevance and ranking results


How does that work?

Boolean search is direct search

We can get the exact results we're looking for if we know how to do boolean search and our database. **The problem is knowing the exact terms for what we're looking for.**

This isn't how we commonly search today or how academic databases present results. They rank by something called relevance.

Academic database and relevancy/ranking features today



HomeProduct Information ▼Learning

DISCOVERY & SEARCH

What is SmartText searching?

🕒 Oct 25, 2018 Knowledge

EBSCOhost and EBSCO Discovery Service offer SmartText Searching, a natural language search strategy that allows you to enter as much text as you like for a search—a phrase, sentence, paragraph, or even whole pages.

How SmartText Searching Works:

1. SmartText Search looks at the selection of text entered into the search box and first runs it through a sophisticated summarizer, identifying the main words/phrases in the text.
2. SmartText Search uses those main words/phrases and queries them against the database, getting back a sorted list of the words/phrase that are most relevant as compared to that database's content.
3. Based on this result, a relevance weight is assigned to each word/phrase.
4. A search string is then built OR'ing the terms and their weights together, and a search is conducted against the database.
5. A relevant Result List is returned.

Note: When SmartText Searching is applied to a search performed from the Advanced Search with **Guided Style** Find Fields screen, the SmartText algorithm is only applied to the term or terms in the topmost find field. Any terms in subsequent find fields use the standard Boolean logic.

Clarivate™ProQuestEx LibrisInnovative

Search...Search

HomeManage CasesSubmit a CaseContact Us ▼Platform Status ▼

Get Link

Share This

Notify Me

Report Article

Save ▼

ProQuest Search Results - Relevance Ranking

Article Number: 000034402

The following parameters influence how results are ranked when you choose to sort by **Relevance**. Please note, this is not a comprehensive list.

- **Exact Term vs Individual Word:** If you intend to search for an exact term rather than separate words, you'll want to put quotes around the term (ie "infectious diseases" instead of infectious diseases). Otherwise, the system looks for each separate and distinct word and documents containing the exact phrase would not necessarily have a higher relevance ranking.
- **Term Frequency:** This is how often search terms appear. Documents that contain search terms more frequently will have an increased relevance ranking.
- **Inverse Document Frequency:** This is how often these terms appear in any document in ProQuest. A term that is more "rare" such as autoimmune will raise the relevancy ranking compared to a term such as "test" which is more common.
- **Metadata Field Weighting:** This changes relevance ranking depending on what field a term appears in. For example, a term appearing as a subject is considered more important than that same term appearing in the full-text.

Open text ranking algorithms

To rank results we need to use some kind of algorithm or formula that returns a number. While we don't know exactly what the academic databases are doing we do know some open methods like:

Set-based algorithms

Represent the query and documents as sets of tokens (with no duplicates) and compare them.

Vector algorithms

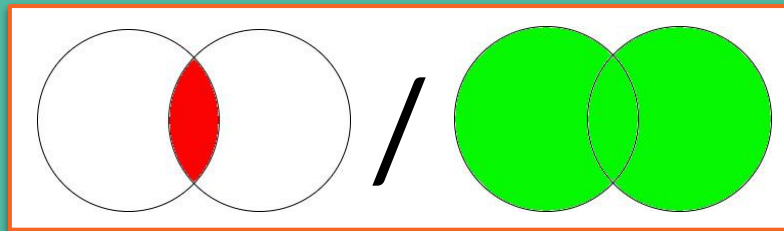
Represent the query and documents as lists of numbers based on word importance and then compares them.

Jaccard Set Similarity example

Sets are ordered groups of data with no duplicates.

To get the set similarity between a query and document(s) we take the intersection of the the query and a document (the OR) and divide by the union (the AND).

Relevancy is determined by the ordered results.



```
# Function to calculate Jaccard Similarity
# incorporating word_matches for partial and wildcard matching
def jaccard_similarity(query_set, doc_set):
    # Calculate the intersection using word_matches for partial matching
    # This is the boolean AND
    intersection = sum(1 for term in query_set if
                       any(word_matches(term, doc_set)
                           for word in doc_set))

    # Calculate the union using word_matches for partial matching
    # This is the boolean OR
    union = len(query_set) + len(doc_set) - intersection

    return intersection / union if union > 0 else 0 # Avoid division by zero
```

Sparse Vectors

Another option is to **vectorize** the documents, which means transforming each one into a mathematical representation (a vector). This converts documents and queries from text into vectors (lists of numbers). It's sparse because most of the columns will be zero.

Document	california	disney	disneyland	fun	has	in	is	rides	world
Disney World is fun.		#		#			#		#
Disneyland is in California.	#		#			#	#		
Disney World has rides.		#			#			#	#

Scoring the vectors

We have a few options to score, AKA assign values, to the vectors. Two of the most basic are:

Bag of Words (BoW)

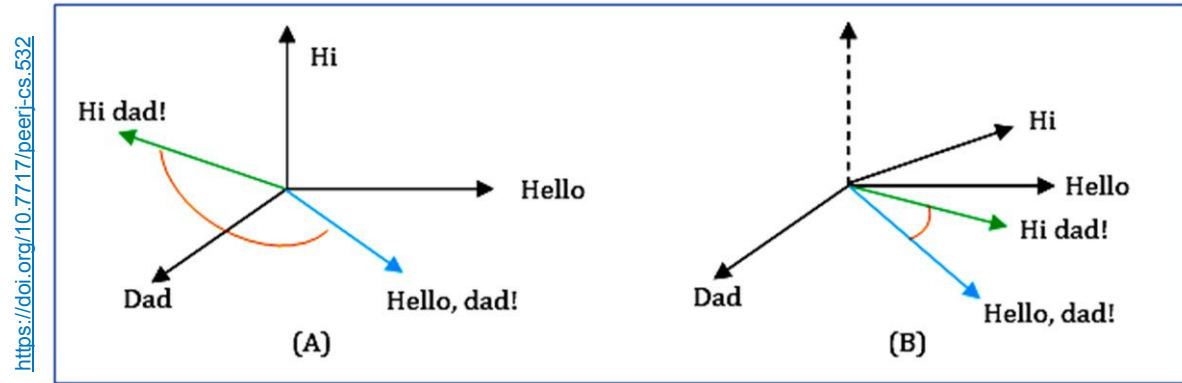
Score each cell with a one when a word is in the document.

Term Frequency

Score each word by the number of times it appears in the document.

Calculating similarity in a matrix

We treat the query and each document as a vector (a list of numbers) in a multi-dimensional space. The number of dimensions (or columns) is the number of unique terms across all documents. We use **cosine similarity** to compare them. This measures the angle between the two document vectors. **If the angle is small, the documents are similar.**



Discussion: How does the inclusion of term frequency change things?

- What happens when looking for a term in a longer document?
 - Would longer documents be ranked higher or lower than shorter ones?
 - What about looking for the term turtle in a group of documents about turtles?
 - How meaningful is the word turtle?
 - What about the inclusion of common but overall meaningless words (aka stopwords) like “the”, “a”, “and”, and “is”?
 - If we’re looking at term frequency how do we handle those?
-

Term Frequency - Inverse Document Frequency (TF-IDF)

This method penalizes the term frequency if it appears throughout the document.

If we have a set of documents about cancer then the word cancer is less meaningful.

Luckily TF-IDF is already implemented in scikit-learn so we don't have to code it from scratch.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Initialize the TF-IDF Vectorizer
vectorizer = TfidfVectorizer()

# Fit and transform the documents into TF-IDF vectors
tfidf_matrix = vectorizer.fit_transform(documents)

# Transform the query into a TF-IDF vector (using the same vectorizer)
query_vector = vectorizer.transform([query])

# Compute cosine similarity between the query and all documents
similarity_scores = cosine_similarity(query_vector, tfidf_matrix)[0]

# Rank documents based on similarity scores
ranked_indices = np.argsort(similarity_scores)[::-1] # Sort in descending order
ranked_scores = similarity_scores[ranked_indices]

# Print ranked results
print("Ranked Documents:")
for i, (idx, score) in enumerate(zip(ranked_indices, ranked_scores)):
    print(f"{i+1}. Score: {score:.4f} - {documents[idx]}")
```

Best Matching 25 (Okapi BM25)

This is a probabilistic model that takes into account term frequency and inverse document frequency but normalizes based on the document length.

Longer documents can be penalized even if they have unique search terms.

BM25 is implemented in a Python library. Like TF-IDF we don't have to write it from scratch.

```
# Initialize the BM25 model
bm25 = BM25Okapi(tokenized_documents)

# Query to search for

# Tokenize the query
tokenized_query = preprocess(query)

# Get the BM25 scores for each document based on the query
scores = bm25.get_scores(tokenized_query)

# Retrieve the documents with the highest BM25 scores
ranked_docs = sorted(enumerate(scores), key=lambda x: x[1], reverse=True)
```

```
# Set parameters for BM25 Okapi
# These are defaults

bm25.k1 = 1.5 # Term frequency saturation
# A higher k1 means that it's more sensitive to
# term frequencies.

bm25.b = 0.75 # Length normalization
# A higher b means that it normalizes less for document length,
# meaning longer documents will be less penalized.
```

Unexamined ranking methods

Probabilistic algorithms

Estimate the **probability** that a document is relevant to a query based on token occurrences.

In essence, what is the likelihood that a document generate the query?

Dense vectors (word embeddings)

These models use a shallow neural network with one hidden layer to calculate vectors for each token based on its **relationship to other tokens within a context window**.

There are **no zero values** in the vectors making it dense, not sparse.

TextRank

TextRank, inspired by Google's PageRank, ranks words or phrases using a **graph-based (network)** approach. Words are nodes, and edges are formed between words that appear close together. Each word's importance is based on its relationships to others in the graph.

Conclusion

Search is not neutral

- Where are the humans in this process?
 - Are we able to determine the programmer's or designer's intent? Why and how could this be hidden from the user?
 - How many times and in how many ways did my decisions affect the results of the ranking algorithms?
 - How could I or a database publisher further affect the results? And to what ends?
 - How can the user determine what to trust when it comes to search and ranking?
-

Future workshops

We have more workshops coming up, some of which can be found at <https://rcfoundations.research.columbia.edu/>

- **Music Technology Workshop Series** (ongoing until the end of the semester)
 - Sponsored by the Gabe M. Weiner Music & Arts Library
- **Using Google Cloud at Columbia: Fundamentals for Researchers** (Mar 26)
 - Sponsored by Research Computing Services - CUIT
- **Getting Credit for your (Computer) Code** (April 9)
 - Sponsored by Columbia University Libraries
- **Level Up Your Lit Review** (April 10)
 - Sponsored by Gottesman Libraries, Teachers College
 - With Ava Kaplan, Research and Instruction Librarian



Thank you
