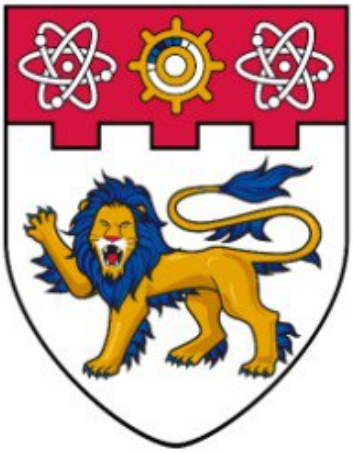
# School of Computer Science and Engineering



NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

## SC4021/CZ4034 - Information Retrieval

AY 2023/2024, Semester 2

## Project Report



| Name | Matriculation Number |
|---|---|
| Cholakov Kristiyan Kamenov | U2123543B |
| Haja Kiyasudeen Nusrath Hajara | U2022440L |
| T. Daranidarran | U2023492H |
| Ma Chunyou | U2023930A |
| Rayudu Abhiteja Phani | U2021983B |
| Wu Rongxi | U2020719J |

# Table of Contents

# 1. Introduction

## 1.1 Background: Opinion Search Engine (OSE)

In today's digital world, there's an abundance of opinions on virtually every topic imaginable. However, navigating this vast sea of information can be daunting for users seeking specific viewpoints or sentiments on particular subjects. To address this challenge, developing an Opinion Search Engine (OSE) with integrated sentiment analysis capabilities emerges as a solution.

The primary goal of this project is to design and implement an OSE that enables users to efficiently find relevant opinions. The system will incorporate sentiment analysis algorithms to classify opinions as positive, negative, or neutral, providing users valuable insights into public sentiment surrounding their queries.

By harnessing the power of natural language processing (NLP) and machine learning techniques, the OSE aims to streamline the process of opinion discovery and analysis. Ultimately, this tool will empower users to make informed decisions, conduct comprehensive research, and gain deeper insights into public sentiment

## 1.2 Selected Topic

The chosen topic for this project is virtual reality (VR) and augmented reality (AR), focusing on AR/VR headsets. This topic was chosen because of its current popularity and intriguing viewpoints. AR/VR technologies, especially headsets, have garnered significant attention in recent years due to their transformative potential across various industries, such as gaming, entertainment, healthcare, and education. The rapidly evolving landscape of AR/VR headsets presents a rich source of diverse perspectives, insights, and discussions.

By focusing on this topic, the project aims to present insights into opinions and sentiments surrounding AR/VR technologies as a whole, as well as specific AR/VR headsets. This analysis will provide valuable information for understanding user perceptions, preferences, and experiences within the AR/VR community on Reddit.

# 2. Data Crawling

Question 1.1: How was the corpus crawled (e.g., source, keywords, API, library) and stored?

## 2.1 Data Source

For this project, Reddit serves as the primary data source. Reddit was chosen due to its ease of use and the availability of the Reddit API, which facilitates convenient access to a wide range of discussions and content. The dataset comprises posts and comments from several relevant subreddits, including:

- r/virtualreality
- r/augmentedreality
- r/VisionPro
- r/MetaQuestVR
- r/oculus
- r/OculusQuest

These subreddits are frequented by enthusiasts, developers, and users of AR/VR technologies, making them ideal sources for capturing diverse opinions, trends, and sentiments within the AR/VR community on Reddit.

## 2.2 Data Crawling

The project utilized the Reddit API with the “Python Reddit API Wrapper” (PRAW) library, its Python wrapper, to efficiently crawl and retrieve the relevant data from the specified subreddits. This involved iterating through the top 1000 submissions of the subreddits (PRAW requests are limited to 1000 at the time of this project), extracting relevant information such as submission title, author, creation time, number of comments, score, and URL (Figure 2.1).

```python
# Browse the submissions
for submission in reddit.subreddit(subreddit_name).top(limit=limit):
    # Print the progress - submission title, submission cnt
    print(f'{subreddit_name}-{submission_cnt}', submission.title)
    # Define the submission
    new_submission = {
    "author": submission.author,
    "created_utc": submission.created_utc,
    "distinguished": submission.distinguished,
    "id": submission.id,
    "name": submission.name,
    "num_comments": submission.num_comments,
    "score": submission.score,
    "selftext": submission.selftext,
    "title": submission.title,
    "upvote_ratio": submission.upvote_ratio,
    "url": submission.url
}
```

**Figure 2.1: Submission Extraction**

It also retrieves comments associated with each submission, capturing details like author, body, creation time, and score (Figure 2.2). These data were stored in lists and then converted into two Pandas DataFrames, one for submissions (representing the main posts) and another for comments.

```python
submission.comments.replace_more(limit=0)
# Browse the comments
for comment in submission.comments.list():
    # Print the progress - comment cnt
    print(f'comment #{comment_cnt}')
    # Define the comment
    new_comment = {
    "author": comment.author,
    "body": comment.body,
    "body_html": comment.body_html,
    "created_utc": comment.created_utc,
    "distinguished": comment.distinguished,
    "id": comment.id,
    "link_id": comment.link_id,
    "parent_id": comment.parent_id,
    "score": comment.score
}
```

**Figure 2.2: Comment Extraction**

The Pandas DataFrames for submissions and comments were then systematically stored in CSV files (in the corresponding directories) for further analysis and processing. This structured storage format (Figure 2.3) ensures easy accessibility and organization of the retrieved information, facilitating subsequent steps in data exploration, visualization, and analysis.

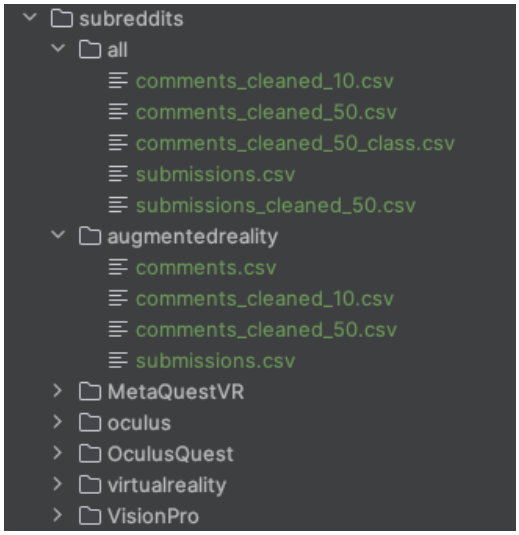



**Figure 2.3: Crawled Data Directories**

## 2.3 Data Cleaning & Analysis

After collecting the Reddit data, it was analyzed to search for missing or duplicated values. Reddit API gives the deleted/removed comments, too, and these comments were spotted by analyzing the most frequent comments, where the removed comments had “body” values: “[deleted]”, “removed”, etc. Also, it was found that several authors were publishing the same comment several times. This is expected because the communities are public, and many users use them for marketing purposes (mainly advertising their AR/VR-related products). Furthermore, the data consisted of many useless records for the project’s purposes. Initially, around 500,000 comments were collected, which were later reduced to 100,000 after the cleaning procedure.

The cleaning (Figure 2.4) consisted of:

- Removing all comments with more than 3 occurrences (consider them spam).
- Ignoring all comments with less than 50 words in their “body” fields, as these comments were observed to contain more meaningful text. Since the crawled dataset was large, there was the flexibility to choose only large records while maintaining 100,000 comments.

- Removing all comments with the same body and author, to avoid duplication of comments.
- Removing comments with a high percentage (>50%) of special characters, these comments were mostly presented by spamming emojis.
- Removing comments with less than 40% unique words, again, these comments were mostly spam of repeating phrases.

The cleaned DataFrame of comments, the result of the mentioned above stages, was used as the comments' corpus for this project. The submissions' corpus was defined by getting only the submissions whose "id" can be found in the cleaned DataFrame.

```
# Remove the comments that occur in all_repeated_comments and have more than 3 occurrences
data["comments"] = data["comments"][~data["comments"]["body"].isin(all_repeated_comments[all_repeated_comments > 3].index)]
print(f"{subreddit_name}: Removed {initial_count - len(data['comments'])} comments that occur in all_repeated_comments and have more than 3 occurrences")
initial_count = len(data["comments"])

# Clear the comments with less than min_word_count words
data["comments"] = data["comments"][data["comments"]["body"].apply(lambda x: len(str(x).split()) >= min_word_count)]
print(f"{subreddit_name}: Removed {initial_count - len(data['comments'])} comments with less than {min_word_count} words")
initial_count = len(data["comments"])

# Remove duplicated comments with same body and author
data["comments"] = data["comments"].drop_duplicates(subset=["body", "author"])
print(f"{subreddit_name}: Removed {initial_count - len(data['comments'])} duplicated comments")
initial_count = len(data["comments"])

# Remove comments with high percentage of special characters
data["comments"] = data["comments"][data["comments"]["body"].apply(lambda x: len([c for c in str(x) if not c.isalnum()]) / len(str(x)) < 0.5)]
print(f"{subreddit_name}: Removed {initial_count - len(data['comments'])} comments with high percentage of special characters")
initial_count = len(data["comments"])

# Remove comments with less than 40% unique words
data["comments"] = data["comments"][data["comments"]["body"].apply(lambda x: len(set(str(x).split())) / len(str(x).split()) > 0.4)]
print(f"{subreddit_name}: Removed {initial_count - len(data['comments'])} comments with less than 40% unique words")
```

**Figure 2.4: Comments Cleaning**

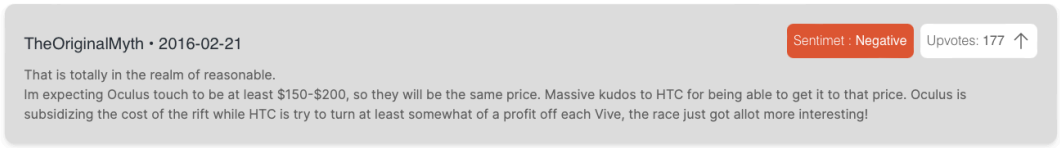
## 2.4 User Applications

### Question 1.2: What kind of information might users like to retrieve from the crawled corpus (i.e., applications) with sample queries?

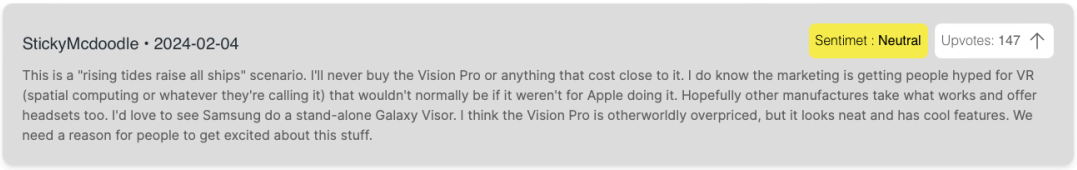
Users may have various interests and objectives when querying from the crawled corpus with the OSE system. They may like to know about general information on AR/VR, perception of the topic, user sentiment towards specific headsets and emerging technology.

Possible queries from the user could be (with corresponding results from the web app):

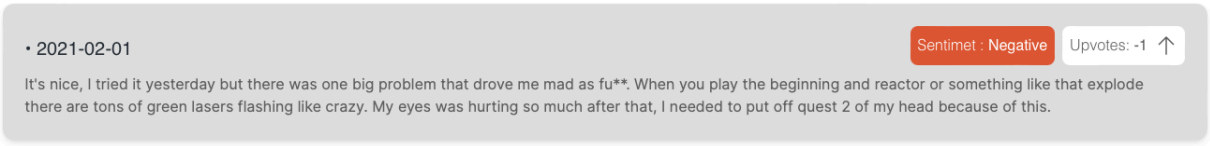
1. "oculus price"



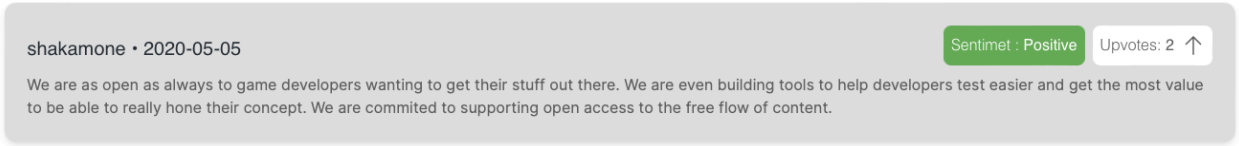

2. “vision pro price”



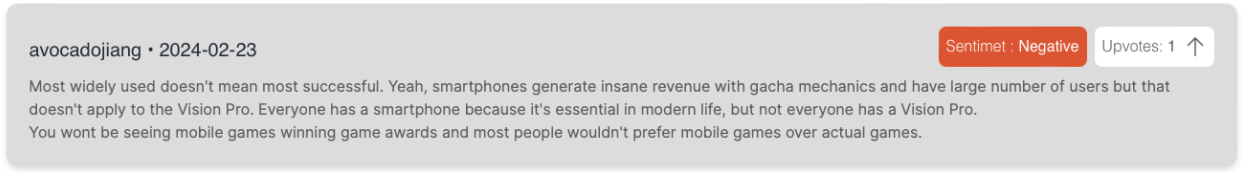

3. “my eyes hurt so much”



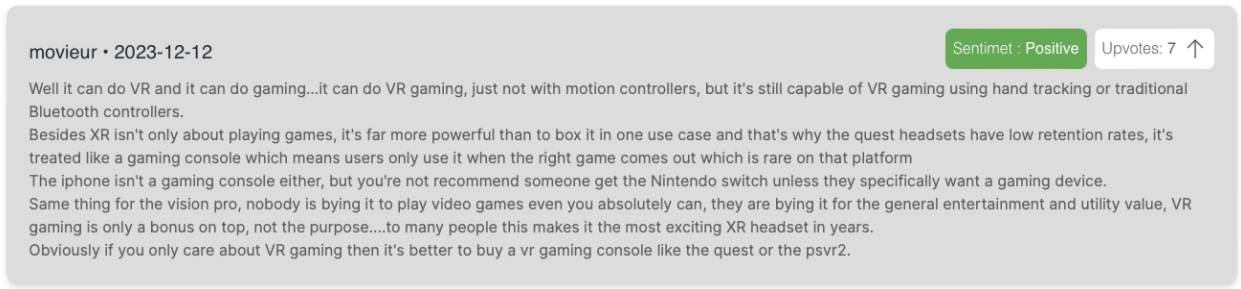

4. “developer tools ar”



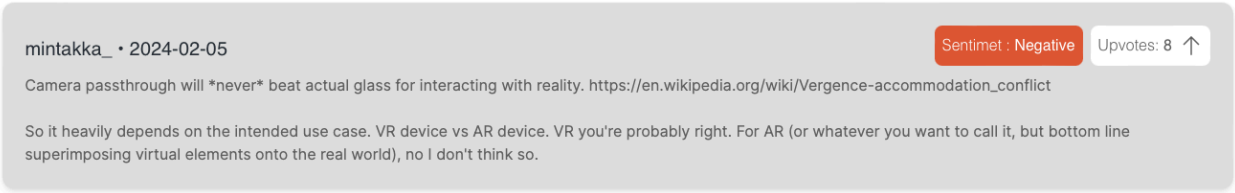

5. “vision pro mobile games”




6. “vr gaming”



7. “ar vs vr”




## 2.5 Corpus Details

Question 1.3: The number of records, words, and types (i.e., unique words) in the corpus.

The corpus is made of 94,130 comments (Table 2.1). In total, it has 9,071,933 words, of which 254,377 are unique, and the results are duplicates. Also, a set of 60,787 synonyms and lemmatization forms was created and used when searching in the corpus. The details of the corpus are presented in Table 2.1 below.

| | |
|---|---|
| Number of Records | 94,130 |
| Number of Words | 9,071,933 |
| Number of Unique Words | 254,377 |
| Number of Unique Synonyms | 60,787 |

**Table 2.1: Corpus Details**

# 3. Text Corpus Indexing and Querying

This project utilizes Apache Solr, an open-source Java-based information retrieval library, to index crawled data. Solr helps to preprocess the documents by tokenizing, applying various linguistic modules and then it indexes the document. Some important Apache Solr features include robust full-text search capabilities, scalability to handle large datasets and traffic, dynamic clustering for organizing search results, and seamless integration with RESTful APIs for easy interaction with other systems.

## 3.1 Selected Data Fields

The following relevant fields and their contents from the scraped data from the subreddits were added to Solr before indexing.

| **Field** | **Description** | **Type** | **Example** |
|---|---|---|---|
| id | Unique identifier for each record | string | ‘kt607bb’ |
| author | Author of the Reddit post | text_general | ‘Trip_b3’ |
| body | Comment left by the author regarding the topic (at least 50 words) | text_general | ‘Agree with the comments would love to play with it but $13 is too expensive. Would pay a couple of dollars and maybe even pay another couple of dollars for each new filter you create to change the room. Enough cool filters and I may have actually paid more than $13, but entry price is just too high. I have a feeling that you would make more money by lowering the price.’ |
| body_html | HTML version of the body | text_general | ‘ <p>Agree with the comments would love to play with it but $13 is too expensive. Would pay a couple of dollars and maybe even pay another couple of dollars for each new filter you create to change the room. Enough cool filters and I may have actually paid more |

| | | | |
|---|---|---|---|
| | | | than $13, but entry price is just too high. I have a feeling that you would make more money by lowering the price.</p><br> ’ |
| created_utc | Comment’s creation timestamp in UTC | pdoubles | 1709483928.0 |
| distinguished | Whether comment is distinguished or not | text_general | false |
| link_id | Submission id that the comment belongs to | text_general | ‘t3_1b5j7u6’ |
| parent_id | ID of the parent comment | text_general | ‘t3_1b5j7u6’ |
| upvotes | Number of upvotes minus number of downvotes on comment | pdoubles | 31.0 |
| subreddit | Subreddit that comment is from | text_general | ‘VisionPro’ |

**Table 3.1: Indexed data details**

## 3.2 Solr Configurations

In Apache Solr, the schema configuration includes the definition of multiple fields, each tailored with specific filters and configurations to meet its unique processing needs. A fundamental aspect of these configurations is the tokenization of text, which is systematically executed according to the defined requirements for each field. Properly configured indexing ensures the precise and efficient processing of documents. This arrangement not only enhances the relevance of search results but also optimizes the overall performance of the search engine. Solr also offers support in about 31 different languages, such as Arabic, Bulgarian, Catalan, Czech, Danish, German, Greek, English, Spanish, Estonian, Basque, Persian, Finnish, French, Irish, Galician, Hindi, Hungarian, Armenian, Indonesian, Italian, Japanese, Latvian, Dutch, Norwegian, Portuguese, Romanian, Russian, Swedish, Thai, and Turkish.

Here is the list of configurations set in Solr:

### 3.2.1 Tokenization techniques

Solr uses various tokenization techniques to break down text fields into searchable terms/tokens. These are the different types of tokenizers used in Solr:

1. Standard tokenizer
    a. Designed to split text into tokens based on the Unicode Text Segmentation algorithm, which is the most common form of breaking up text into words.
2. Whitespace tokenizer
    a. Designed to split text into tokens based on whitespace characters such as spaces, tabs and line breaks.

The standard tokenization is configured to be applied in 31 different languages.

Tokenization is crucial for indexing as it transforms the raw text into elements that can be indexed and searched. Each term becomes an entry in the search index, allowing the search engine to quickly locate documents containing the terms. Furthermore, through tokenization, a search engine can support various types of searches, including exact matches, partial matches and phrase searches.

### 3.2.2 Stopword removal

Removes common words in the 31 different languages.

Common words include words that occur frequently but carry little individual meaning. This is to save indexing size, which can lead to faster search performance because there are fewer terms to scan during a query. It also helps the search engine to focus on the more meaningful words in the text, leading to more relevant search results.

Benefits of Stopword Removal:

- **Reduced Index Storage Size:** Removing common words (stopwords) from indexing processes reduces the size and complexity of the search index. This leads to faster indexing times and a more streamlined search database, improving overall system performance.
- **Enhanced Search Speed:** By excluding stopwords, which are generally irrelevant to the meaning of queries, the search engine has fewer terms to process. This results in quicker query execution and faster response times for users.
- **Improved Relevance of Search Results:** Excluding common stopwords helps to focus the search on the more meaningful words within the user queries. This

can improve the relevance of the search results, as the matching process concentrates on the terms that truly affect the content's meaning.

### 3.2.3 Character normalization

Converts text to a standard and consistent form only in the following languages:

- Arabic
    - Uses arabicNormalization filter.
- German
    - Uses germanNormalization filter.
- Persian
    - Uses arabicNormalization and persianNormalization filters.
- Hindi
    - Uses indicNormalization and hindiNormalization filters.

Normalization reduces the complexity of the search space, and different forms of the same word are treated as one.
Benefits of Character Normalization:

- **Enhanced Search Accuracy:** Character normalization transforms text into a consistent format by standardizing variations of characters. This consistency helps in matching queries accurately to the indexed documents, improving the precision of search results.
- **Reduced Index Storage Size:** By reducing the variety of character representations, normalization decreases the complexity of the index. This approach allows for more efficient storage and quicker retrieval processes, as fewer variations of each word need to be stored.
- **Broader Search Coverage:** Normalization ensures that different representations of characters do not hinder search effectiveness. For example, converting ligatures or diacritical marks to their basic forms expands the scope of search queries to include more variations, thus capturing a broader range of documents.

### 3.2.4 Lowercasing

Lowercasing is applied to all languages.

Lowercasing is the process of converting all letters in a text to their lowercase form. This standardization technique reduces the lexical diversity of text data, ensuring that the search engine treats the variations of the same word equally.

**Benefits of Lowercasing:**

- **Enhanced Search Consistency:** Lowercasing text in queries and documents ensures that the search engine treats different case variations of the same word uniformly. This standardization significantly enhances the consistency of search results, enabling users to find relevant information regardless of text case.
- **Reduced Index Storage Size:** By converting all text to lowercase, Solr can reduce the number of unique terms stored in the index. This reduction helps in minimizing the storage space required for the index, improving search performance.

### 3.2.5 Stemming

Stemming is applied to 31 different languages. The Snowball Porter algorithm is used to apply the stemming.

The Snowball Porter algorithm is a method used to reduce words to their base or root form, known as stems. This algorithm helps streamline various forms of a word to a common stem. It applies a set of rules to strip common morphological and inflectional endings from words. It is designed to be both fast and conservative, it avoids excessive modification of words, thereby maintaining the integrity of the original terms.

Benefits of stemming:

- **Enhanced Search Consistency:** Stemming normalizes variations of words to their base form, thereby enabling the search engine to match more documents that contain any form of the stemmed word. This consistency significantly improves the comprehensiveness of search results.
- **Reduced Index Storage Size:** By reducing words to their root forms, stemming decreases the overall size of the search index. This reduction not only speeds up the indexing process but also makes querying more efficient, as there are fewer distinct terms to evaluate.
- **Broader Search Coverage:** Stemming allows users to retrieve documents that contain derivatives of a search term, not just the exact term. This flexibility enhances the relevance of search results, accommodating the variations in the content.

### 3.2.6 Phonetic Encoding

The Double Metaphone algorithm is applied to words to convert them into a phonetic code for the English language.

This allows for indexing and retrieval of text by how it sounds rather than spelt. This technique is particularly useful in search systems to accommodate variations in spelling, especially for names or terms with multiple correct or common spellings.

### 3.2.7 Synonym Expansion

Synonym expansion associates different terms with similar meanings within a search system to enhance its understanding and responsiveness to user queries. This technique involves its synonyms, broadening the search's scope.

To achieve this, we first need to get the synonym dictionary.

```
comments_synonyms = {}

for word in comment_terms:
    comments_synonyms[word] = []
    for syn in wordnet.synsets(word):
        for lemma in syn.lemmas():
            comments_synonyms[word].append(lemma.name())

# remove the ones with empty list values
comments_synonyms = {k: v for k, v in comments_synonyms.items() if v}
```

And import it into the Solr server.

```
url = "http://localhost:8983/solr/new_core/schema/analysis/synonyms/en"
payload = json.dumps(synonyms)
headers = {'Content-type': 'application/json'}
response = requests.request("PUT", url, headers=headers, data=payload)
```

Afterward, we configured a field type <sgm_text_en> and added the SynonymGraphFilterFactory synonym filter to this field. Subsequently, we changed the field type of the fields to be searched to this new field type. During the search, the synonym filter retrieves the corresponding synonyms and generates search results.

```
<fieldType name="sgm_text_en" class="solr.TextField" multiValued="false">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.ManagedSynonymGraphFilterFactory" managed="en"/>
    <filter class="solr.FlattenGraphFilterFactory"/>
    <filter class="solr.TrimFilterFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="lang/stopwords_en.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.ASCIIFoldingFilterFactory" preserveOriginal="true"/>
    <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.TrimFilterFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="lang/stopwords_en.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.ASCIIFoldingFilterFactory" preserveOriginal="true"/>
    <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
  </analyzer>
</fieldType>
```

Synonym expansion is only available for queries in the English language.

**Benefits of Synonym Expansion:**

- **Broader Search Coverage:** Synonym expansion allows the search engine to consider multiple terms that have similar meanings, expanding the scope of search queries. This approach ensures that users can retrieve relevant documents even if they don't use the exact terms present in the content.
- **Enhanced Search Accuracy:** By mapping synonyms to key terms, Solr enhances the accuracy of search results. This feature helps users who may use different terminology to refer to the same concept.

### 3.2.8 Spell Checking

Spell checking is an essential feature in Solr that enhances the search experience by identifying and correcting misspellings in search queries. In Solr, spell checking is implemented through a spell checker component that can be configured to run during both indexing and querying processes.

**Implementation of Spellchecking:**

**Spellchecker Configuration:**

Solr uses a spell checker component that is defined in the solrconfig.xml file. This configuration includes setting up a specific dictionary the spell checker will use to validate and suggest corrections. The dictionary is built from the indexed data itself.

**Building the Dictionary:**

The dictionary is built using the terms from the indexed documents. A special query (with the query parameter: q=*:*&spellcheck=true&spellcheck.build=true) triggers Solr to build or rebuild the spellcheck dictionary for all terms in the index. This ensures that the dictionary is comprehensive and includes all possible variations and common misspellings of words found in the documents.

**Query Time Spell checking:**

If the spellcheck parameter is true during querying, Solr checks each term in the user's query against the dictionary. If a misspelling is detected, Solr suggests the most likely correction based on the dictionary. This is particularly useful for improving user interaction, as users are either automatically provided with corrected search results or suggestions for correct spellings.

**Benefits of Spell checking:**

- **Improved Search Accuracy:** By correcting misspellings in search queries, Solr helps reduce zero-result pages, thereby improving the accuracy of the search results.
- **Enhanced User Experience:** Offering spelling corrections and suggestions helps in maintaining an engaging user experience, especially in handling typographical errors which are common in user queries.
- **Increased Search Flexibility**: Spell checking makes the search engine more robust against input errors, particularly in a dynamic search environment with diverse data and user inputs.

By effectively handling misspellings, Solr ensures that the search functionality remains robust, accurate, and user-friendly.

## 3.3 Indexing steps

Steps to index documents in Solr

1. Download Java and Solr
2. Prepare the database records

3. Upload the CSV file through the Solr admin UI
4. managed-schema.xml file is generated with all the relevant indexing and processing configurations.
5. Configurations can be upgraded and changed in the Solr admin UI.

## 3.4 Data Querying

Querying data from a Solr index involves constructing a request URL that specifies the search criteria and configuration parameters tailored to optimize search accuracy and efficiency. Below, we discuss the essential components and functionalities employed in constructing a query URL.

**Base URL and Core Selection:**

The base URL for querying in Solr is structured to point to a specific Solr core that contains the indexed data. For example:

*http://localhost:8983/solr/${solr_core_name}/select*

This URL specifies the new_core Solr core and the select handler, which processes search queries:

*http://localhost:8983/solr/new_core/select*

**Essential Query Parameters:**

1. **Query Specification (q):**

This parameter is fundamental in defining what to search for within the index. It requires specifying the field and the search term, formatted as field:term. For instance, searching for "automatic" within the "body" field of documents is represented as body:automatic. The term needs to be URL-encoded to ensure the web server correctly interprets it. This is how a query parameter would look like:

*?q=body:automatic*

2. **Spell Checking (spellcheck):**

Set to true, this parameter enables Solr's spell-checking feature, which is vital for providing user-friendly search experiences, especially in handling typographical errors in query inputs. It helps in suggesting or automatically correcting misspelled terms within the search queries. This is how a URL would look like:

*&spellcheck=true*

3. **Output Format (wt):**

This parameter specifies the format of the search results. By setting wt=json, the response from Solr is formatted as JSON, a lightweight and easy-to-parse data format, which is particularly beneficial for web-based applications where JavaScript can readily manipulate such data. This is how a URL would look like:

*&wt=json*

4. **Building the Spellcheck Dictionary (spellcheck.build):**

Used in conjunction with the spellcheck parameter, setting this to true instructs Solr to build or rebuild the spellcheck dictionary. This process is crucial for keeping the dictionary up-to-date with the latest terms in the index, ensuring the spellcheck feature remains effective over time. This is how a URL would look like:

*&spellcheck.build=true*

Users can perform the following search query to retrieve relevant documents. Since Solr provides RESTful APIs for querying documents, users can query for documents using GET requests. This is particularly useful when displaying queries on the application's front end.

## Question 2.2: Write five queries, get their results, and measure the speed of the querying

These are some of the queries that users can request on the Solr server:

1. **Normal Query:** Querying "visionpro" results in Solr returning all results containing the query term.

API called to Solr server:

*http://localhost:8983/solr/comments/select?indent=true&q.op=AND&q=body%3A%22visionpro%22&useParams=*

The q parameter in the URL is set to body:visionpro meaning that the user is requesting to search for records that contain the word 'visionpro' in the body field.

**Time taken to run this query: 8 ms**

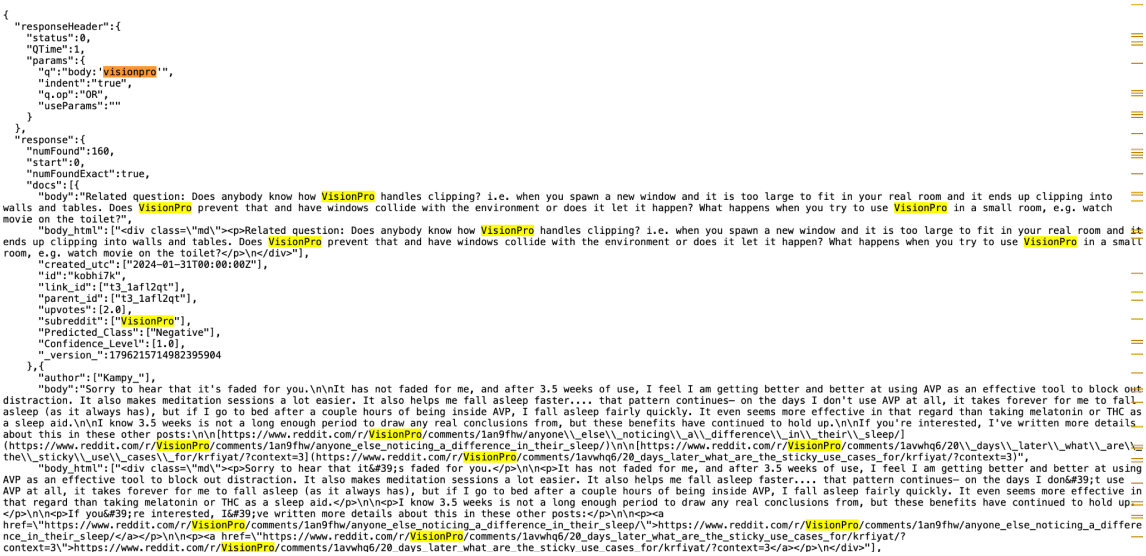



**Figure 3.1: Query "visionpro" partial JSON result**

As you can see from the above figure through the highlighted words, the JSON response returns a list of dictionaries with every record that contains 'visionpro' in the body field.

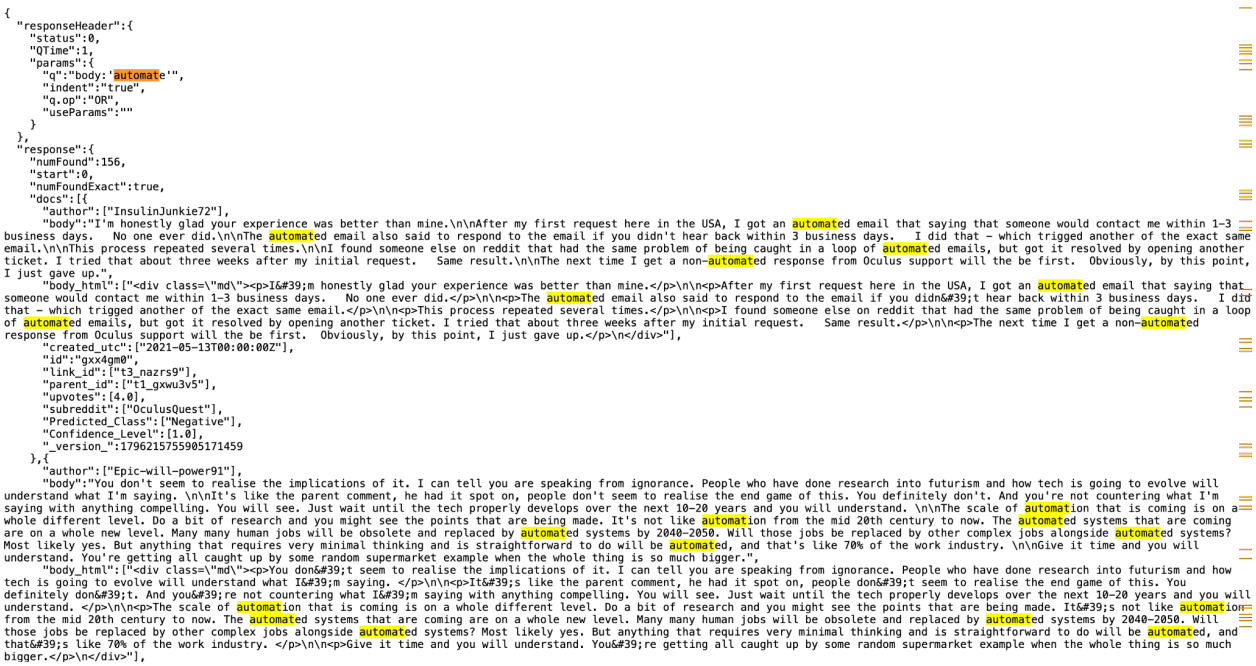
2. **Stemming:** Querying "automate" results in Solr returning all results containing the query term. It also returns documents containing "automated", "automation", etc. as a result of stemming of the query term.

API called to Solr server:

*http://localhost:8983/solr/comments/select?indent=true&q.op=OR&q=body%3A%27automate%27&useParams=*

The q parameter in the URL is set to body:automate meaning that the user is requesting to search for records that contain the word automate and the associated words through stemming in the body field.

**Time taken to run this query: 32 ms**



```
{
  "responseHeader":{
    "status":0,
    "QTime":1,
    "params":{
      "q":"body:'automate'",
      "indent":"true",
      "q.op":"OR",
      "useParams":""
    }
  },
  "response":{
    "numFound":156,
    "start":0,
    "numFoundExact":true,
    "docs":[{
      "author":["InsulinJunkie72"],
      "body":"I'm honestly glad your experience was better than mine.\n\nAfter my first request here in the USA, I got an automated email that saying that someone would contact me within 1-3
business days.   No one ever did.\n\nThe automated email also said to respond to the email if you didn't hear back within 3 business days.   I did that - which trigged another of the exact same
email.\n\nThis process repeated several times.\n\nI found someone else on reddit that had the same problem of being caught in a loop of automated emails, but got it resolved by opening another
ticket. I tried that about three weeks after my initial request.   Same result.\n\nThe next time I get a non-automated response from Oculus support will the be first.  Obviously, by this point,
I just gave up.",
      "body_html":["<div class=\"md\"><p>I&#39;m honestly glad your experience was better than mine.</p>\n\n<p>After my first request here in the USA, I got an automated email that saying that
someone would contact me within 1-3 business days.   No one ever did.</p>\n\n<p>The automated email also said to respond to the email if you didn&#39;t hear back within 3 business days.   I did
that - which trigged another of the exact same email.</p>\n\n<p>This process repeated several times.</p>\n\n<p>I found someone else on reddit that had the same problem of being caught in a loop
of automated emails, but got it resolved by opening another ticket. I tried that about three weeks after my initial request.   Same result.</p>\n\n<p>The next time I get a non-automated
response from Oculus support will the be first.  Obviously, by this point, I just gave up.</p>\n</div>"],
      "created_utc":["2021-05-13T00:00:00Z"],
      "id":"gxx4gm0",
      "link_id":["t3_nazrs9"],
      "parent_id":["t1_gxwu3v5"],
      "upvotes":[4.0],
      "subreddit":["OculusQuest"],
      "Predicted_Class":["Negative"],
      "Confidence_Level":[1.0],
      "_version_":1796215755905171459
    },{
      "author":["Epic-will-power91"],
      "body":"You don't seem to realise the implications of it. I can tell you are speaking from ignorance. People who have done research into futurism and how tech is going to evolve will
understand what I'm saying. \n\nIt's like the parent comment, he had it spot on, people don't seem to realise the end game of this. You definitely don't. And you're not countering what I'm
saying with anything compelling. You will see. Just wait until the tech properly develops over the next 10-20 years and you will understand. \n\nThe scale of automation that is coming is on a
whole different level. Do a bit of research and you might see the points that are being made. It's not like automation from the mid 20th century to now. The automated systems that are coming
are on a whole new level. Many many human jobs will be obsolete and replaced by automated systems by 2040-2050. Will those jobs be replaced by other complex jobs alongside automated systems?
Most likely yes. But anything that requires very minimal thinking and is straightforward to do will be automated, and that's like 70% of the work industry. \n\nGive it time and you will
understand. You're getting all caught up by some random supermarket example when the whole thing is so much bigger.",
      "body_html":["<div class=\"md\"><p>You don&#39;t seem to realise the implications of it. I can tell you are speaking from ignorance. People who have done research into futurism and how
tech is going to evolve will understand what I&#39;m saying. </p>\n\n<p>It&#39;s like the parent comment, he had it spot on, people don&#39;t seem to realise the end game of this. You
definitely don&#39;t. And you&#39;re not countering what I&#39;m saying with anything compelling. You will see. Just wait until the tech properly develops over the next 10-20 years and you will
understand. </p>\n\n<p>The scale of automation that is coming is on a whole different level. Do a bit of research and you might see the points that are being made. It&#39;s not like automation
from the mid 20th century to now. The automated systems that are coming are on a whole new level. Many many human jobs will be obsolete and replaced by automated systems by 2040-2050. Will
those jobs be replaced by other complex jobs alongside automated systems? Most likely yes. But anything that requires very minimal thinking and is straightforward to do will be automated, and
that&#39;s like 70% of the work industry. </p>\n\n<p>Give it time and you will understand. You&#39;re getting all caught up by some random supermarket example when the whole thing is so much
bigger.</p>\n</div>"],
```


**Figure 3.2: Query "automate" partial JSON result with stemming**

As you can see from the above figure through the highlighted words, the JSON response returns a list of dictionaries with every record that contains 'automate' and its associated stemming words in the body field.

3. **Spell checking:** Querying "automtic" results in Solr returning no results containing the query term. However, Solr is able to spell check and suggest an alternate similar word for this query.

API called to Solr server:

*http://localhost:8983/solr/comments/select?indent=true&q.op=OR&q=body%3A%22automtic%22&spellcheck.build=true&spellcheck.reload=true&spellcheck=true&useParams=*

The q parameter in the URL is set to body:automtic meaning that the user is requesting to search for records that contain the word 'automtic'. Since the spellcheck parameter is also set to true, it checks for similar words to 'automtic' and suggests the correct spelling of the words to the user.

**Time taken to run this query: 6 ms**

```
{
  "responseHeader":{
    "status":0,
    "QTime":28,
    "params":{
      "q":"body:'automtic'",
      "indent":"true",
      "spellcheck":"true",
      "q.op":"OR",
      "spellcheck.reload":"true",
      "spellcheck.build":"true",
      "useParams":""
    }
  },
  "command":"build",
  "response":{
    "numFound":0,
    "start":0,
    "numFoundExact":true,
    "docs":[ ]
  },
  "spellcheck":{
    "suggestions":["automtic",{
      "numFound":1,
      "startOffset":6,
      "endOffset":14,
      "suggestion":["automatic"]
    }]
  }
}
```


**Figure 3.3: Query "automtic" JSON result**

4. **Synonym expansion:** Querying "small" results in Solr returning results containing either "small" or its synonyms such as "little", "tiny", etc.

API called to Solr server:

*http://localhost:8983/solr/comments/select?indent=true&q.op=OR&q=body%3A%27small%27&useParams=*

The q parameter in the URL is set to body:small, meaning that the user is requesting to search for records that contain the word 'small' or its synonyms.

**Time taken to run this query: 38 ms**

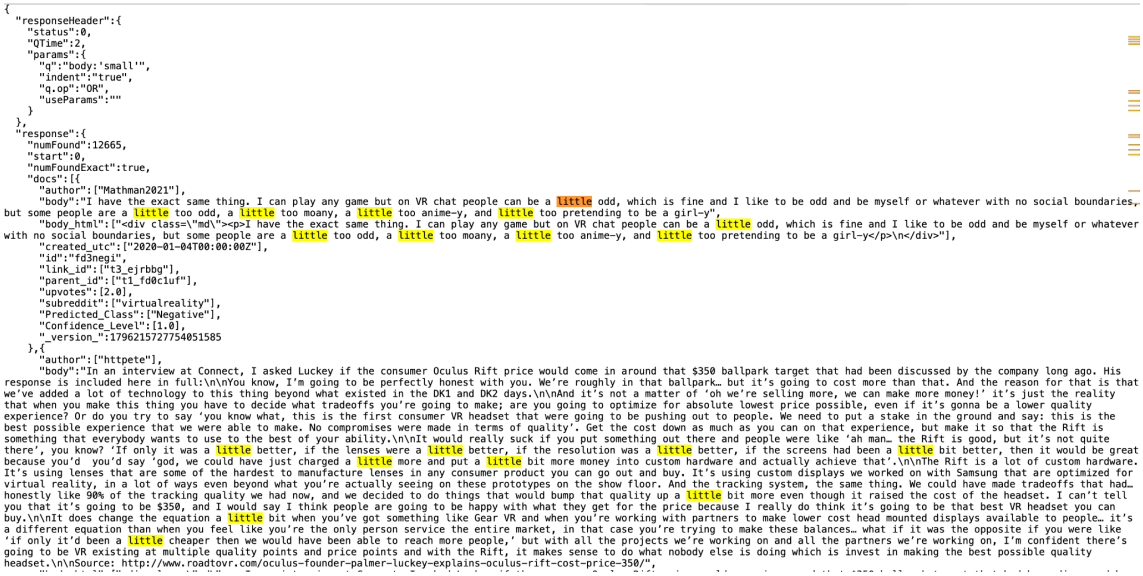



**Figure 3.4: Query "small" partial JSON result with synonym expansion**

As can be seen from the figure above, a JSON response is populated with a list of dictionaries of records that contain 'small' or its synonyms, such as 'little', which is highlighted in the figure.

5. **Phrase Proximity:** Querying "automate process" ~5 results in Solr returning all results with phrases that contain at most 5 words between 'automate' and 'process'.

API called to Solr server:

*http://localhost:8983/solr/comments/select?indent=true&q.op=OR&q=body%3A%27automate%20process%27~5&useParams=*

The q parameter in the URL is set to body:"automate process"~5, meaning that the user is requesting to search for records that contain the word 'automate' and 'process' within 5 words of each other.

**Time taken to run this query: 94 ms**

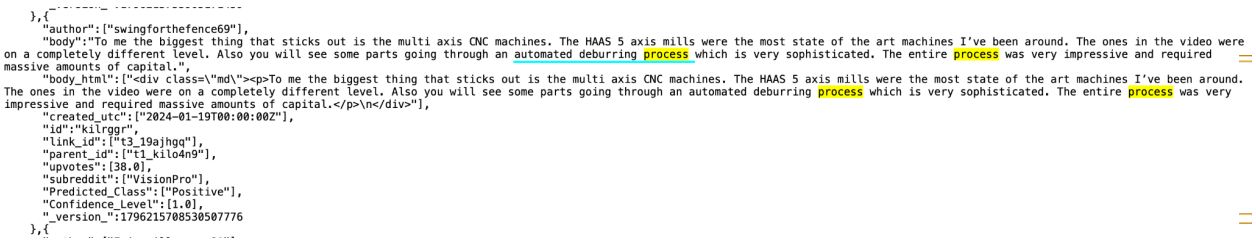



**Figure 3.5: Query "automate process"~5 partial JSON result**

As can be seen in the figure above, a JSON response is populated with a list of dictionaries with records that contain the words ‘automate’ and ‘process’ within 5 words of each other, as shown by the underlined phrase.

6. **Timeline search:** Query a time range to retrieve results from Solr for a specific time period. Querying results between 2022-01-01 to 2023-01-01:

API called to Solr server:

*http://localhost:8983/solr/comments/select?indent=true&q.op=OR&q=created_utc%3A%5B2022-01-01T00%3A00%3A00Z%20TO%202023-01-01T00%3A00%3A00Z%5D&useParams=*

The q parameter in the URL is set to created_utc:created_utc:[2022-01-01T00:00:00Z TO 2023-01-01T00:00:00Z], meaning that the user is requesting to search for records that are created between 2022-01-01 to 2023-01-01.

**Time taken to run this query: 35 ms**


```
{
  "responseHeader":{
    "status":0,
    "QTime":2,
    "params":{
      "q":"created_utc:[2022-01-01T00:00:00Z TO 2023-01-01T00:00:00Z]",
      "indent":"true",
      "q.op":"OR",
      "useParams":""
    }
  },
  "response":{
    "numFound":13048,
    "start":0,
    "numFoundExact":true,
    "docs":[{
      "author":["ToureTaylor"],
      "body":"Imagine Amazon making a vr shopping app that simulates real life grocery stores. The shopping app will have real life products. They'll have aisles based on what catagory the
product is in (just like real life). When you put something in your cart in the vr app, it'll mimick the amazon cart from their website when you click add to cart. The cart in the vr app will
display a total price of everything in your cart. At check out, your required to input your shipping address and payment method. Once your done checking out, the items that are bagged will go
to a vr simulated truck displaying your address and est. arrival time. I'm in deep thoughts right now.",
      "body_html":["<div class=\"md\"><p>Imagine Amazon making a vr shopping app that simulates real life grocery stores. The shopping app will have real life products. They&#39;ll have aisles
based on what catagory the product is in (just like real life). When you put something in your cart in the vr app, it&#39;ll mimick the amazon cart from their website when you click add to
cart. The cart in the vr app will display a total price of everything in your cart. At check out, your required to input your shipping address and payment method. Once your done checking out,
the items that are bagged will go to a vr simulated truck displaying your address and est. arrival time. I&#39;m in deep thoughts right now.</p>\n</div>"],
      "created_utc":["2022-01-08T00:00:00Z"],
      "id":"hrql713",
      "link_id":["t3_rybbln"],
      "parent_id":["t3_rybbln"],
      "upvotes":[0.0],
      "subreddit":["virtualreality"],
      "Predicted_Class":["Positive"],
      "Confidence_Level":[0.9999],
      "_version_":1796215726963425281
    },{
      "author":["suckyurmother"],
      "body":"I really want a fully fleshed out game where you can go into different invironments that can be stressful or unpleasant like a grocery store or an office for example and you can
just destroy shit freely and fight the people there, just to tap into the rage that I get from these places",
      "body_html":["<div class=\"md\"><p>I really want a fully fleshed out game where you can go into different invironments that can be stressful or unpleasant like a grocery store or an office
for example and you can just destroy shit freely and fight the people there, just to tap into the rage that I get from these places</p>\n</div>"],
      "created_utc":["2022-01-08T00:00:00Z"],
      "id":"hrqi2z4",
      "link_id":["t3_rybbln"],
      "parent_id":["t3_rybbln"],
      "upvotes":[1.0],
      "subreddit":["virtualreality"],
      "Predicted_Class":["Neutral"],
      "Confidence_Level":[0.8936],
      "_version_":1796215726964473856
    },
```


**Figure 3.6: Query timeline range partial JSON result**

As can be seen in the figure above, the JSON response contains only a list of dictionaries with a creation date that lies between the range that is queried.

# 4. User-Interface(UI) for Searching (Web Application)

Question 2.1: Design a simple UI to allow users to access your system in a simple way.

This section introduces the UI part of the project - the web application. The web application is used to make the search experience user-friendly and bypass the usage of the Apache Solr server.

## 4.1 Used Technologies and Tools

The web application was built using the Next.js framework, Next.js is a popular framework extending React. However, it combines front-end React components with server-side rendering. The components are still React ones but have the benefit of bein pre-rendered on the server side. The latest “App Router” structure was adopted for Next.js. The difference between it and “Page Router” is that instead of files directly represent routes, relies on file-based routing where nested folders define routes.

For the purely front-end features, Tailwind CSS and Material Tailwind were introduced. Tailwind was chosen because of its better flexibility, responsiveness and customization options, compared to Bootstrap.

Lastly, the web application utilizes both the front-end design and the back-end intermediate connection between the application and the Apache Solr server.

## 4.2 Pages Design

The application consists of 2 main pages:

- **Main page:** the initial page where the user can use all search features and browse through the pages of results, the initial page is also used for presenting the results of the query.
- **Record page:** the record page is used for making the link between the comment and its submission.

The application follows a unified design characterized by a minimalistic and clean aesthetic. It uses white space to create a sense of simplicity and cleanness.

### 4.2.1 Main Page

The main page (before the search is executed) is what the users will see at first. They are welcomed by a cheering animated logo and a simplistic modern design. As seen in Figure 4.1, it consists of 2 main functional components: the search bar and the filters section.

The first component is the search bar, it is built of an input field and an icon button for executing the search. There is also a hidden select input for spellcheck suggestions. The second main component is our filters section, where the user can apply filters about the subreddit of the comment, its upvotes range, publish date range, sentiment prediction and the comment's author. This whole section can be shown and hidden by pressing the filters button.

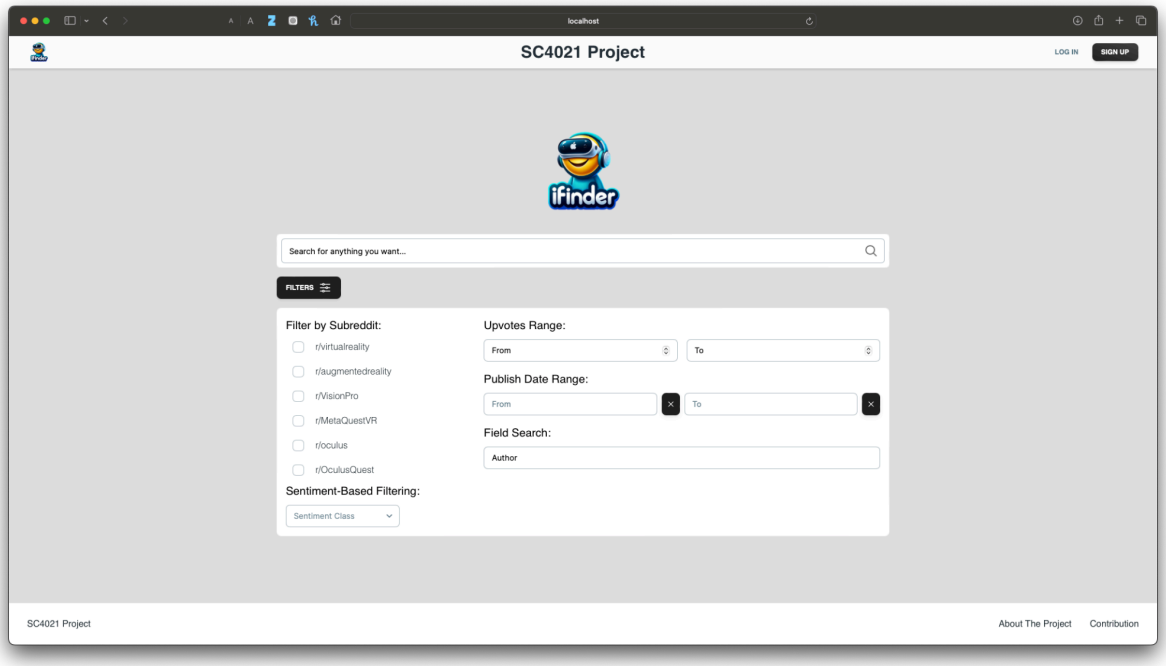



**Figure 4.1: Home Page Design**

### 4.2.2 Main Page with Search Executed

In Figure 4.2, a change in the design of the main page is observed after a query has been executed. Following are the 3 main components of the UI when the search is executed.

The first and most important new section is the container of the results from the query. It consists of a single-row container with the sorting functionality and a container for the search results. The sorting functionality is presented by a select input where the user can select between sorting by recommendation, upvotes or publish date. Next to it,

there is an icon button about the direction of the sorting: ascending or descending. In the results section, all results are represented in their own card holding the subreddit it is extracted from, their body content, their author, upvotes score and publish date, the predicted sentiment class, and a button that redirects the user to the record page, where the comment is linked with its submission (post).

The other 2 added components are the left and right containers, holding the plots that display the distribution of the subreddits of the results (on the left) and the distribution of the predicted sentiment class of the comments (on the right). The plots are displayed using React Plotly.js, which utilizes React components with the functionality of the original Plotly.js library. Below the left plot, a field has been implemented to hold the number of results from the current query.

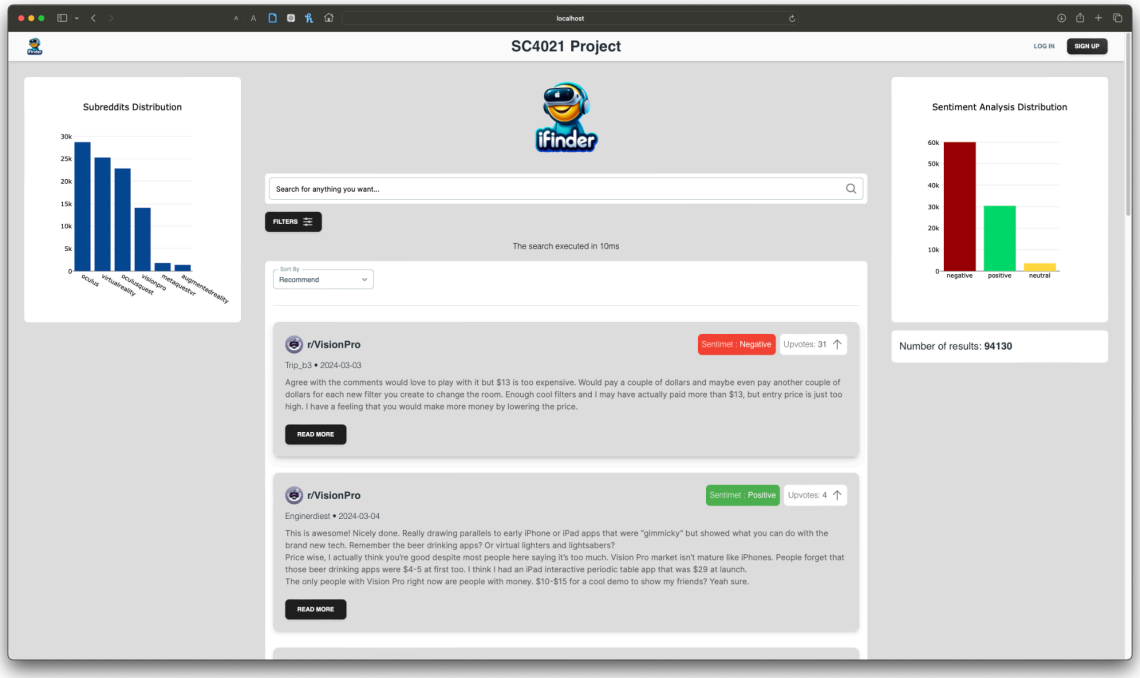



**Figure 4.2: Home Page with Search Executed - Design**

### 4.2.3 Record Page

The record page simply connects the comment and the submission (post) it was extracted from. As seen in Figure 4.3, the submission data (subreddit, author, publish date, title, selftext, upvotes score) and a link to the post in Reddit are displayed.

As for the comment, all the duplicated elements with the submission, like the subreddit, are removed and added below the submission with an indentation on the left to show the difference between the submission and the comment and give a response-like feeling.

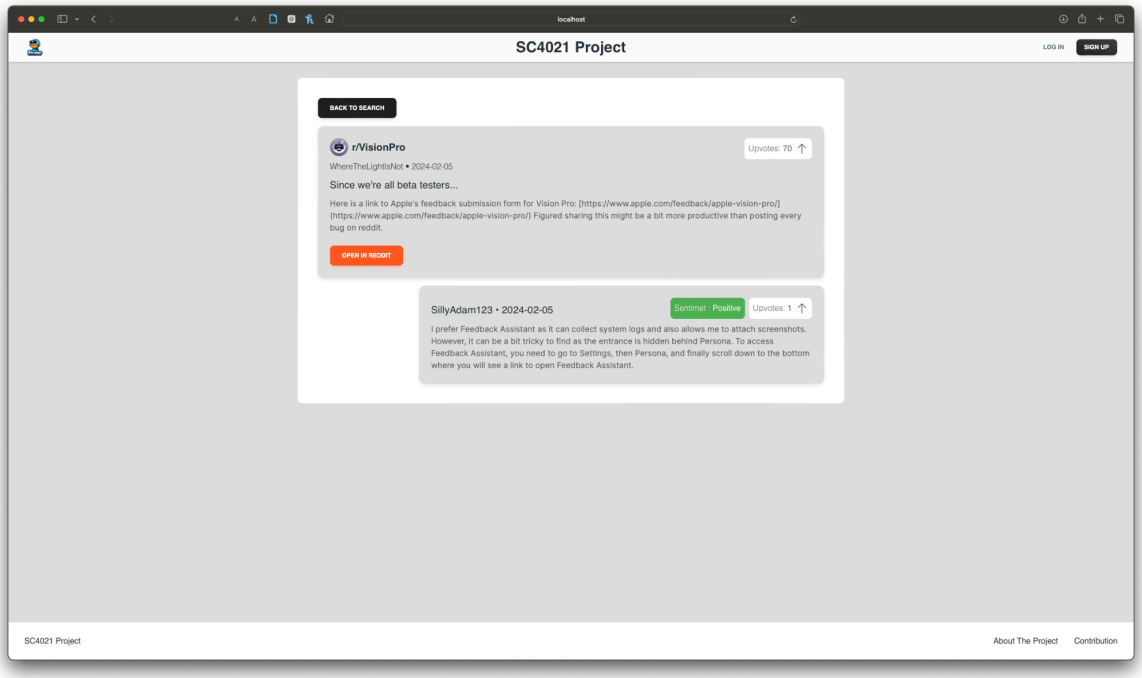



**Figure 4.3: Record Page Design**

## 4.3 Elements Functionality

This section will discuss how the pages' individual elements (components) contribute to the web application's overall functionality and how useState (React Hook allowing state variables across the pages) changes variables in our application. These useState variables are later used when requesting the results from the Apache Solr server by first sending a middleware request to the API backend of the application.

### 4.3.1 Search Bar

The search bar sets the useState of the comment's "body" query and triggers a request when the icon button is clicked.

### 4.3.2 Filters Section

As seen in Figure 4.4, the filters section contains all the inputs for the filters we have implemented in our web application.

The **filtering by subreddit** is implemented via a series of checkboxes, this enables selecting multiple subreddits. When each checkbox is clicked, the value of the subreddit is added or removed from an array useState variable. When enabled, the user will get the results of all ticked subreddits.

The **sentiment filtering** is a select input where the user can select the predicted class of the comments. When enabled, only comments with the selected class will be retrieved.

The **upvotes** and **publish date range filtering** are utilized via two inputs of the corresponding type (number or date). When one value is selected, the other one will automatically be treated as the upper/lower limit, and only the results in the interval will be shown. Only comments within the range will be retrieved when both values are set.

The **author filter** is simply an input that returns the selected author's comments.

All filters can be used in any combination, and each can be used individually by simply interacting with it and ignoring the others.

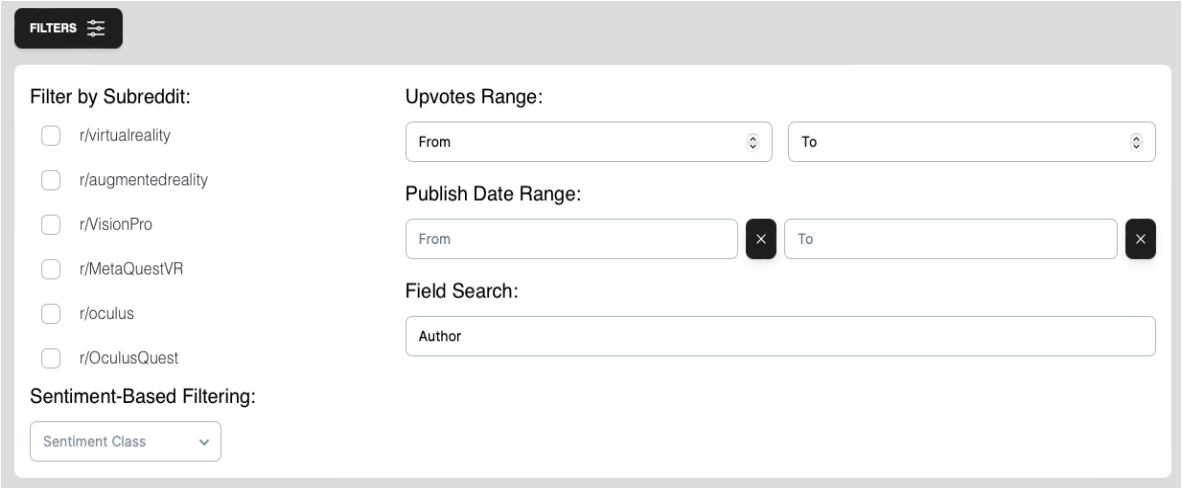



**Figure 4.4: Filters Section**

### 4.3.3 Spell Check Suggestion

The spell check suggestion select appears only after the search is executed, and the Apache Solr server detects that there is a possible spelling error. The select will show all possible spelling suggestions and, after clicking on one will immediately use it as an input for the search bar.

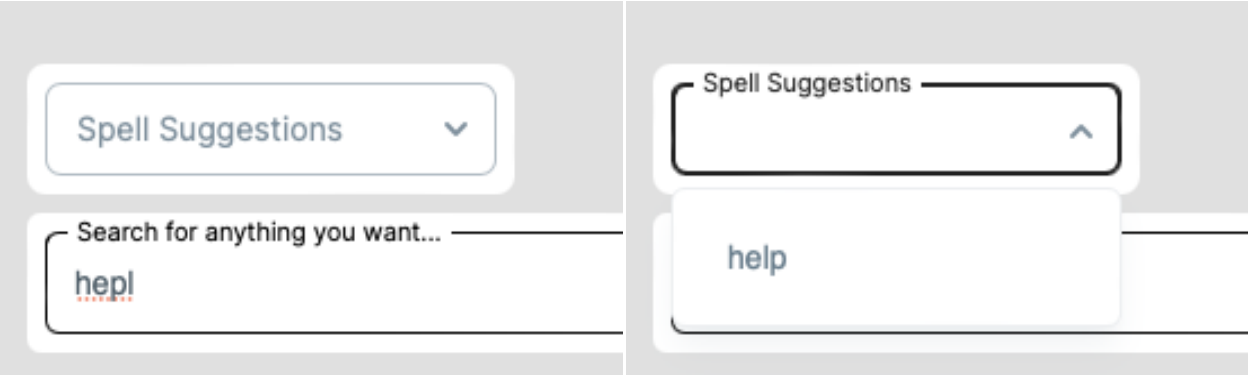


**Figure 4.5: Spell Check Suggestion Select**

### 4.3.4 Sorting Options

One select input and an icon button implement the sorting options (Figure 4.6.). The select button determines the field by which the results are sorted and the icon button determines the direction (ascending/descending) of the sorting. The default value is “Recommend” and is simple sorting by the default Solr sorting from the query. Again, sorting is independent and can be combined with any other filtering/searching combination.

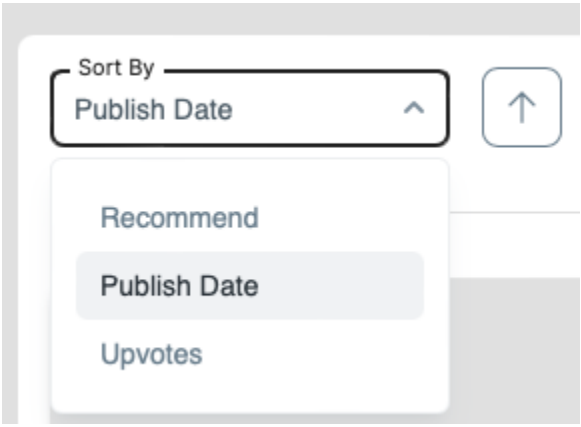



**Figure 4.6: Sorting Options**

### 4.3.5 Paging

The paging component (Figure 4.7.) allows us to perform faster requests and queries from the Solr server because we are limiting the results to be retrieved to only the number of records on a single page, which in our case is set to be 10. The paging comprises the page number and the buttons for the previous/next page around it. A new request is sent to the Solr server on each page click to retrieve the next range of comments.


**Figure 4.7: Paging**

Question 3: Explore some innovations for enhancing indexing and ranking and explain their importance in solving specific problems with examples.

### 4.3.6 Distribution Plots

The subreddits and sentiment class prediction distribution plots (Figure 4.8.) are interactive Plotly.js plots containing the distributions of all the results from the query (not only the page results). They are reloaded on each search and include Plotly.js tools like saving the plots to an image file, zooming them, etc.

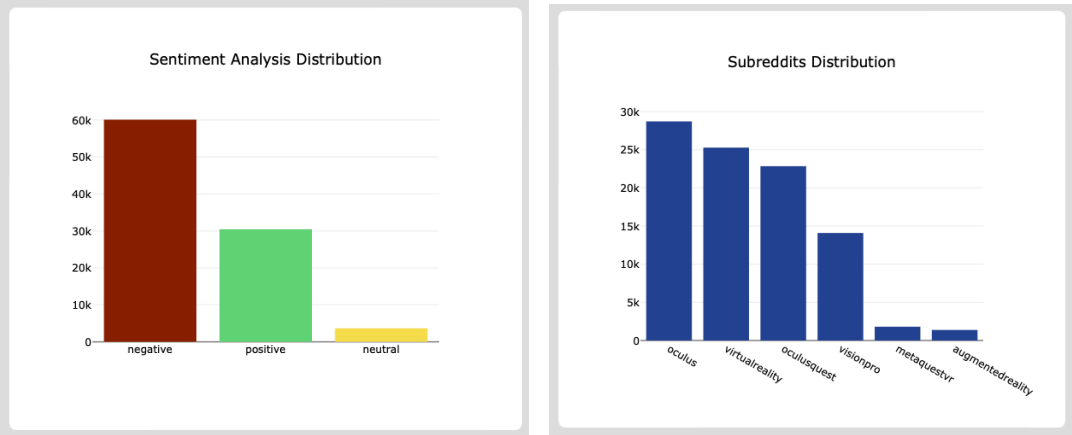



**Figure 4.8: Distribution Plots**

### 4.3.7 Results Counter

As seen in Figure 4.9, the results counter simply holds the number of all results for each query and is reloaded on every search.

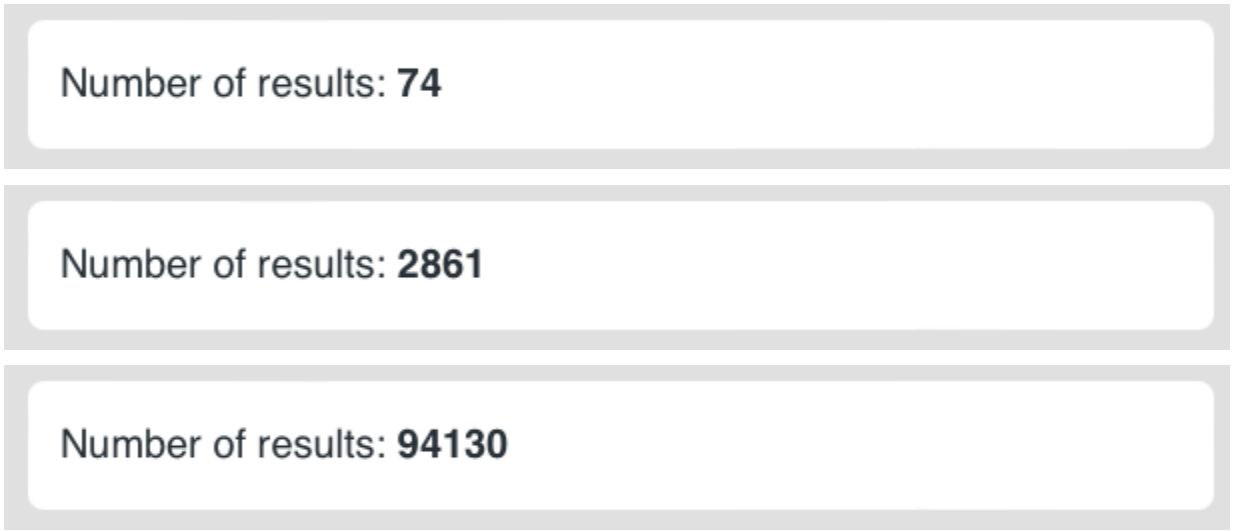



**Figure 4.9: Results Counter**

## 4.4 Backend Integration

In this section, we will discuss the backend implementation for the pages and what requests the Next.js API backend (the middleware in the project) sends to the Apache Solr server. The Next.js API backend sends requests from the frontend to the route **"/api"**. This implementation is better than directly sending the requests from the frontend because the user cannot see the request passed to the Solr server, the user sees only what they are supposed to see.

### 4.4.1 Searching Functionality

Performing a search query is done by clicking the search bar icon button or changing the sorting options. When this occurs, all necessary useState variables are sent to the Next.js API backend via the route **"/api/search"**.

In the backend, the query is being constructed, our query is dynamic and has 3 stages of trying to return results. The first stage is searching with the applied filters and the query **body:"example query"~5**, this query tries to find a match of the query phrase where the individual terms occur within 5 words from each other. If this query gets less than 10 results, a second request to the Solr server will be sent with an updated query **body:"example query"~20**, this one will again try to match the phrase but now the terms from the query phrase can be 20 words apart. Lastly, if the second query fails to get enough queries, the last request will be sent with an updated query where it will search for a phrase consisting of at least one of the terms, the query will look like **body:"term1" OR body:"term2" OR … OR body:"termN"**. This is implemented to maintain strongly related query results, and try weaker if there are not enough results.

The implementation of the body query composition in the figures below.


**Figure 4.10: 1st Attempt Body Query**


**Figure 4.11: 2nd Attempt Body Query**

```
query = '';
let terms = squery.split(' ');
for (let i :number  = 0; i < terms.length; i++) {
  query += `${field}:"${terms[i]}"`;
  if (i !== terms.length - 1) {
    query += ' OR ';
  }
}
```

**Figure 4.12: 3rd Attempt Body Query**

Now, proceeding with constructing the sorting part of the query. There are 2 parts of the sorting query: field to sort by and direction. The field and the direction are added to &sort= to construct the query. As a result, the query is **&sort=field direction**.

```
let sortQuery :string  = '';
if (sortField !== '') {
  sortQuery = `&sort=${sortField} ${sortDirection}`;
}
```

**Figure 4.13: 3rd Sorting Query**

A query has to be constructed for each filter to apply the filters. Then they must be added to the search body query with **AND** operations between.

The **author filter** and the **sentiment class filter** are constructed simply using the queries: **author:"exampleAuthor"** and **Predicted_Class="exampleClass"**. The implementation of the 2 filters can be seen in Figure 4.14.

```
if (filters.author !== '') {
  query += ` AND (author:"${filters.author}")`;
}

if (filters.sentiment !== '') {
  query += ` AND Predicted_Class:"${filters.sentiment}"`;
}
```

**Figure 4.14: Author & Sentiment Class Filter Queries**

The **subreddits filter** query is constructed by performing **OR** operation between all **subreddit:"exampleSubreddit"** queries. So the final query looks like this: **subreddit:"s1" OR … OR subreddit:"sN"**. The backend implementation is as follows in Figure 4.15.

```
if (filters.subreddit.length > 0) {
  let subredditQuery = filters.subreddit
    .map((subreddit: string) : string  => `subreddit:${subreddit}`)
    .join(' OR ');
  query += ` AND (${subredditQuery})`;
}
```

**Figure 4.14: Subreddits Filter Queries**

Lastly, the **upvotes range** and the **publish date range filter** queries are implemented as **upvotes:[number1 TO number2]** and **created_utc:[date1 TO date2]**. Where any number or date can be set to * if they are not defined. The backend implementation of the construction of the queries can be seen in Figure 4.16. and Figure 4.17.

```
let upvotesQuery : string  = '';
if (filters.upvotes[0] !== null && filters.upvotes[1] !== null) {
  upvotesQuery += `upvotes:[${filters.upvotes[0]} TO ${filters.upvotes[1]}]`;
} else {
  if (filters.upvotes[0] !== null) {
    upvotesQuery += `upvotes:[${filters.upvotes[0]} TO *]`;
  }
  if (filters.upvotes[1] !== null) {
    upvotesQuery += `upvotes:[* TO ${filters.upvotes[1]}]`;
  }
}
if (upvotesQuery !== '') query += ` AND (${upvotesQuery})`;
```

**Figure 4.16: Upvotes Range Filter Queries**

```
let dateQuery : string  = '';
if (filters.publishDate[0] !== '' && filters.publishDate[1] !== '') {
  dateQuery += `created_utc:[${filters.publishDate[0]}T00:00:00Z TO ${filters.publishDate[1]}T00:00:00Z]`;
} else {
  if (filters.publishDate[0] !== '') {
    dateQuery += `created_utc:[${filters.publishDate[0]}T00:00:00Z TO *]`;
  }
  if (filters.publishDate[1] !== '') {
    dateQuery += `created_utc:[* TO ${filters.publishDate[1]}T00:00:00Z]`;
  }
}
if (dateQuery !== '') query += ` AND (${dateQuery})`;
```

**Figure 4.17: Publish Date Range Filter Queries**

The query has to be encoded before sending the whole search request (Figure 4.18).

```
let encodedQuery : string  = encodeURIComponent(query);
```

**Figure 4.18: Encode The Query**

Finally, our request will look like this:

**http://localhost:8983/solr/comments/select**

**?facet.field=Predicted_Class&facet.field=subreddit&facet=true**

**&q=${encodedQuery}&rows=${numRows}**

**&start=${(page - 1) * numRows}${sortQuery}&spellcheck=true&wt=json**

Enabling spell check with including **&spellcheck=true**.

### 4.4.2 Comment Retrieval

The comment retrieval functionality is used in the record page when only the single selected comment is retrieved. This is done by first sending a request with the id of the comment from the frontend to the backend via the route **"/api/[id]/comment"**, then the backend will send the following request to the Solr server:

**http://localhost:8983/solr/comments/select**

**?q=${encodedQuery}&rows=${1}&fl=*&wt=json**

Where **encodedQuery** is defined as in Figure 4.19.


**Figure 4.19: Comment encodedQuery**

### 4.4.3 Submission Retrieval

For retrieving the submission, we do the same thing, but this time we send the **parent_id** of the comment to the backend via the route **"/api/[id]/submission"**, then the backend will proceed with the following request to the Solr server:

**http://localhost:8983/solr/submissions/select**

**?q=${finalQuery}&rows=${1}&wt=json**

This time we are using our second core **submissions** where all the submissions are stored. Also, the **finalQuery** is defined as follows in Figure 4.20.

```
let id : string  = params.id;
let encodedQuery : string  = encodeURIComponent(id.substring(3, id.length));
let field : string  = 'id';

let finalQuery : string  = `${field}:${encodedQuery}`;
```

**Figure 4.20: Submission finalQuery**

# 5. Classification

## 5.1 Classification Approach

Question 4.1: Motivate the choice of your classification approach in relation to the state of the art.

The classification task utilized a specialized RoBERTa-base architecture extensively pre-trained on approximately 124 million tweets from January 2018 to December 2021. The sheer volume and specificity of the data ensure that the model has a nuanced understanding of the language used on social media, particularly Twitter, which is crucial for accurate sentiment analysis within this context.

In recognizing the intrinsic differences between user behaviors and linguistic patterns on Reddit, the platform from which our data was collected, and Twitter, the platform on which the model was trained, it became imperative to further refine our model's capabilities to ensure its effectiveness across a broader spectrum of social media channels. To achieve this, we leveraged the rich and diverse dataset of IMDb reviews for an additional layer of fine-tuning. This strategic choice not only enhances the model's versatility but also its adaptability to the unique nuances of language and sentiment expression found in different online communities. By incorporating insights gleaned from IMDb reviews, renowned for their depth of user sentiment and complexity of feedback, our model now boasts an unparalleled sensitivity to varying contexts and sentiment expressions. Moreover, this fine-tuning has significantly improved the model's performance on Reddit data, ensuring more accurate and insightful sentiment analysis tailored to the platform's nuances.

The model’s fine-tuning with the TweetEval benchmark, a well-regarded dataset for Twitter sentiment analysis, further refines its ability to discern and classify sentiments into negative, neutral, and positive. This fine-tuning process adapts the model to the intricacies of sentiment analysis, leveraging the broad linguistic understanding gained during pre-training to excel in the specific task of sentiment classification.

RoBERTa Base stands as the state-of-the-art in natural language processing (NLP), widely adopted across academia and industry. Its foundation in the TimeLMs paper, a seminal work in NLP research, underscores its cutting-edge design and methodologies. RoBERTa Base's efficacy in sentiment analysis on Twitter data is well-documented as a benchmark in research publications. Leveraging RoBERTa Base ensures alignment with the latest advancements in NLP, promising robust and insightful analyses in social media sentiment analysis.

Question 4.2: Discuss whether you had to preprocess data (e.g., microtext normalization) and why.

In this project, data preprocessing was crucial in ensuring optimal model performance and adherence to input constraints. The preprocessing steps involved several key processes:

1. **Dropping Comments Without Main Text:** Comments lacking main text ('body') were removed from the dataset. This ensured that only meaningful data containing valuable information for classification tasks was retained.

2. **Sanitizing Text:** Illegal characters such as &, #, ;, {, and } were replaced with colons (:). This sanitization process was particularly important when testing with other models to ensure compatibility and consistency in text representation.

3. **Lowercasing, Punctuation Removal, and Whitespace Stripping:** These standard text normalization techniques were employed when experimenting with sarcasm detection based on the "jkhan447/sarcasm-detection-RoBerta-base" model. Lowercasing ensured uniformity in text representation, while punctuation removal and whitespace stripping helped clean the text for more accurate analysis.

4. **Handling Token Limit:** Comments exceeding the token limit of 512 imposed by the RoBERTa model were disregarded. This decision was critical to prevent input data from exceeding the model's constraints, optimizing computational efficiency and enhancing classification performance.

The input data was effectively cleaned, formatted appropriately, and compliant with the RoBERTa model's constraints by implementing these preprocessing steps. This meticulous preprocessing ensured that the model received high-quality input, leading to more effective and accurate classification outcomes

## 5.2 Evaluation

Question 4.3: Build an evaluation dataset by manually labelling 10% of the collected data (at least 1,000 records) with an inter-annotator agreement of at least 80%.

Manual labeling was done for around 1200 comments (see Figure 5.1). The average Cohen's Kappa between the 3 annotators is 0.867 (see Table 5.1), which is acceptable for inter-judge agreement.

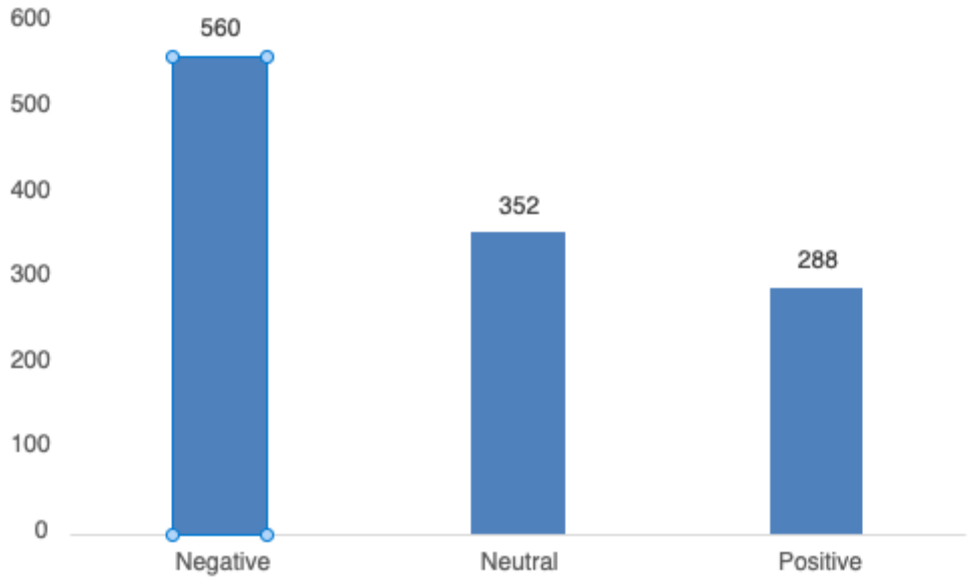



**Figure 5.1. Sentiment Split for Manually Labeled Data (1200 comments)**

| Annotators | Cohen's Kappa |
|---|---|
| Annotator 1 and Annotator 2 | 0.896 |
| Annotator 1 and Annotator 3 | 0.895 |
| Annotator 2 and Annotator 3 | 0.810 |

**Table 5.1: Kappa measure of Inter-Judge Agreements**

Question 4.4: Provide evaluation metrics such as precision, recall, and F-measure on such datasets.

F1, Precision and Recall scores were used to evaluate the ability of 2 classifiers to make predictions. Precision score refers to the measure of positive prediction accuracy. Recall score refers to the measure of a classifier's ability to identify all relevant instances. F1 score summarizes a classifier's ability to correctly predict both positive and negative instances while considering the trade-off between Precision and Recall. We had calculated a decent F1-Score of 0.71351 (see Table 5.2).

The following formulas help calculate a model's F1, Precision and Recall score.

$$Precision = \frac{TruePositive}{TruePositive+FalsePositive}$$

$$Recall = \frac{TruePositive}{TruePositive+FalseNegative}$$

$$F1 = \frac{2*Precision*Recall}{Precision+Recall}$$

| Precision | Recall | F1 score |
|---|---|---|
| 0.667 | 0.767 | 0.71351 |

**Table 5.2: Ensemble Classifier Sentiment  Classification Results**

## Question 4.5: Perform a random accuracy test on the rest of the data and discuss the results.

A subset of the dataset (150 comments) was randomly selected and was ensured to represent the entire dataset in terms of sentiment distribution. The ensemble classifier model was applied to this selected subset of data. The selected data subset were also manually labeled. The model's predictions were compared to the manual labels. Following table 5.3 shows the results of the accuracy of the model's predictions.

| Class | Prediction Accuracy |
|---|---|
| Positive | 0.751 |
| Negative | 0.774 |
| Neutral | 0.738 |

**Table 5.3: Ensemble Classifier Random Accuracy Results**

The model can predict positive and negative (opinionated) comments more accurately than neutral comments. This is probably because the difference between opinionated and neutral comments is blurred and would depend on the context of the comment. However, if more training data were manually labeled, it might have helped with more accurate predictions, as the training data used was extracted from IMDB review comments, Rotten Tomatoes review comments and X tweets.

## Question 4.6: Discuss performance metrics, e.g., records classified per second and scalability of the system.

Documents for this AR/VR subtopics were relatively long, with approximately 96.38 words per comment, resulting in lower classification speeds at 102 records classified per second.

BERT classifier system can be effectively scaled to handle larger datasets, improve training efficiency, and deploy robustly in production environments. Several approaches

to achieve this are model compression (compressing BERT models makes it more memory efficient and fast to deploy) and leveraging distributed training (models can benefit from parallelising training, thereby efficiently utilizing computing resources). However, considering constraints related to computing resources and time limitations, these strategies were not implemented in the current model implementation.

## 5.3 Enhanced Classification

### Question 5: Explore implemented innovations for enhancing classification

The implemented innovation for the classifier was the using ensemble classification. Ensemble classification involves combining the predictions of multiple individual classifiers to produce a final prediction. This approach often leads to better performance than using a single classifier, as it leverages the strengths of different models and mitigates their weaknesses.

The Ensemble classifier was implemented by combining 3 RoBERTa classifiers trained using different datasets (IMDB review comments, X tweets, and Rotten Tomatoes review comments). These datasets were selected as they closely resemble the characteristics often found in Reddit comments. They include long-form opinions, structured feedback, informality, and a mix of sentiments. By leveraging these datasets, we aim to capture Reddit discussions' diverse linguistic styles and expressive nuances. This approach allows our models to better understand and evaluate the multifaceted nature of Reddit comments, ultimately enhancing their ability to generalize and perform effectively.

The ensemble's final prediction was made by combining the individual predictions through a voting mechanism. The document will be considered neutral if there is a three-way tie in the voting process.

# Appendix: Links to Submissions

**Video Presentation:**

https://youtu.be/P68TdXkUfHA

**Datasets, Queries & Results:**

Main Link:

https://entuedu-my.sharepoint.com/:u:/g/personal/kristiya001_e_ntu_edu_sg/EaP-MwF3o8pIpSCfKoNX2PsBpHxsezP-5Ve9jTBqdL_SIQ?e=LyPMx8

Open-source datasets used:

- https://huggingface.co/datasets/carblacac/twitter-sentiment-analysis
- https://huggingface.co/datasets/rotten_tomatoes
- https://huggingface.co/datasets/stanfordnlp/imdb

**Source Codes & Libraries:**

Main Link:

https://entuedu-my.sharepoint.com/:u:/g/personal/kristiya001_e_ntu_edu_sg/EYsuUKU3KdlOocEq7wrWCTMBaWR6hR-1t7rSuNZ9U4-OEA?e=5t690K

Open-source model used:

https://huggingface.co/nlptown/bert-base-multilingual-uncased-sentiment