Report

Table of Contents

Introduction

[TBC]

Literature Review

Following paragraphs are the literature review around the chosen topic.

Full-Text Search Engine

Full-Text search engines perform linguistic searches against text data in target document or collection of text data based on the rules of a particular language (Microsoft Docs, 2020). Full-Text search engines consider the basic unit of Full-Text search as a token rather than a word (Melton & Buxton, 2006). Full-Text queries can include simple words and phrases or multiple forms of a word or phrase. A Full-Text query returns any documents that contain at least one match (also known as a hit). A match occurs when a target document contains all the terms specified in the Full-Text query, and meets any other search conditions, such as the distance between the matching terms. It widely used in modern technology like web search(Google Search, Bing Search, etc.), desktop search(search for local files) and even email search.

As the need of semantic-aware Full-text search and indexing against databases and documents keeps increasing in the past decade, full-text search systems became available to no-expert users who aren't familiar with documents they are searching, don't know the exact vocabulary used to index relevant documents, or don't know how to formulated advanced search queries (Tekli, Chbeir, et al., 2018). All of those issues mentioned above result in noisy or irrelevant search results and false positives & false negatives in the search result (See Precision and Recall).

Precision and Recall

Precision and recall are often used to measure the performance of information retrieval or extraction systems, precision (aka. positive prediction value) is the fraction of relevant results among all results returned (in other words, how many returned results are relevant?), while recall (aka. sensitivity) is the fraction of the amount of returned relevant results among all relevant results in the dataset (in other words, how many relevant results are returned?) (Makhoul, Kubala, et al., 1999) (see Appendix a.).

In general, there is a trade-off between "precision" and "recall". High precision means that fewer irrelevant results are returned as a result of the query (fewer false positives), while high recall means that fewer relevant results are missing (fewer false negatives). traditional text-matching systems such as the LIKE operator in SQL gives you 100% precision with no concessions for recall. A full text search facility gives you a lot of flexibility to tune down the precision for better recall (Erickson, 2008).

For example, the SQL LIKE operator can be extremely inefficient. If you apply it to an un-indexed column, a full scan will be used to find matches (just like any query on an un-indexed field). If the column is indexed, matching can be performed against index keys, but with far less efficiency than most index lookups. Therefore, most full text search implementations use an "inverted index" system to improve performance (see Inverted Index) (Erickson, 2008).

Inverted Index

Inverted Index is an index where the keys are individual terms, and the associated values are sets of records that contain the term. Full text search is optimized to compute the intersection, union, etc. of these record sets, and usually provides a ranking algorithm to quantify how strongly a given record matches search keywords (Erickson, 2008).

As mentioned earlier, non-expert users of full-text search system often don't know how to formulate advanced queries or don't know the vocabularies used when indexing the documents (Tekli, Chbeir, et al., 2018). To address said issue the authors designed and constructed a extended version of inverted index called $SemIndex$, which empowered standard inverted index with a semantic network and a general keyword query model, in order to yield semantic-aware results (ibid.).

Morphological Analysis (Stemming)

[TBC]

Ranking

[TBC]

Semantic Search

"Semantics" refers to the concepts or ideas conveyed by words, and semantic analysis is making any topic (or search query) easy for a machine to understand. [TBC]

Apache Tika

Tika was formally a sub-project of Apache Lucene. Lucene is a Java library for indexing, searching. Lucene also has many other powerful features (e.g. spellchecking, analysis/tokenization, etc.) (Apache Lucene, 2020).

Apache Tika is an open-source toolkit that detects and extracts metadata and text from over a thousand different file types (such as PPT, XLS, and PDF). All of these file types can be parsed through a single interface, making Tika useful for search engine indexing, content analysis, translation, and much more. You can find the latest release on the download page. Please see the Getting Started page for more information on how to start using Tika (Apache Tika, 2020).

DocFetcher

DocFetcher is an existing open-source desktop search application written in Java programming language and Apache Tika framework (see Apache Tika). DocFetcher provides very powerful searching capabilities and as well as a wide range of support document formats (DocFetcher, 2020).

Gantt Chart

The planning of this project is done by utilizing Gantt chart (See Appendix b.).

There're several complications result in a shift of priority to develop a functional cross-platform full-text search API which return search result either from CLI (command-line interface) or `JSON` via REST Web API, instead of building both cross-platform UI and API.

The main processes of this project were requirements analysis, implementation, testing and reporting.

Rationale

This project idea was formed based on specific needs for customisable full-text searching against a huge amount of local documents, and was then inspired by the open-source desktop search application - DocFetcher (see DocFetcher).

As powerful as DocFetcher is, it lacks the full-support for macOS 64-bit platform, also has some minor performance issues. Same thing goes with other existing desktop search applications. So, I decided to build a local document search engine to address said issues and to fit my specific needs.

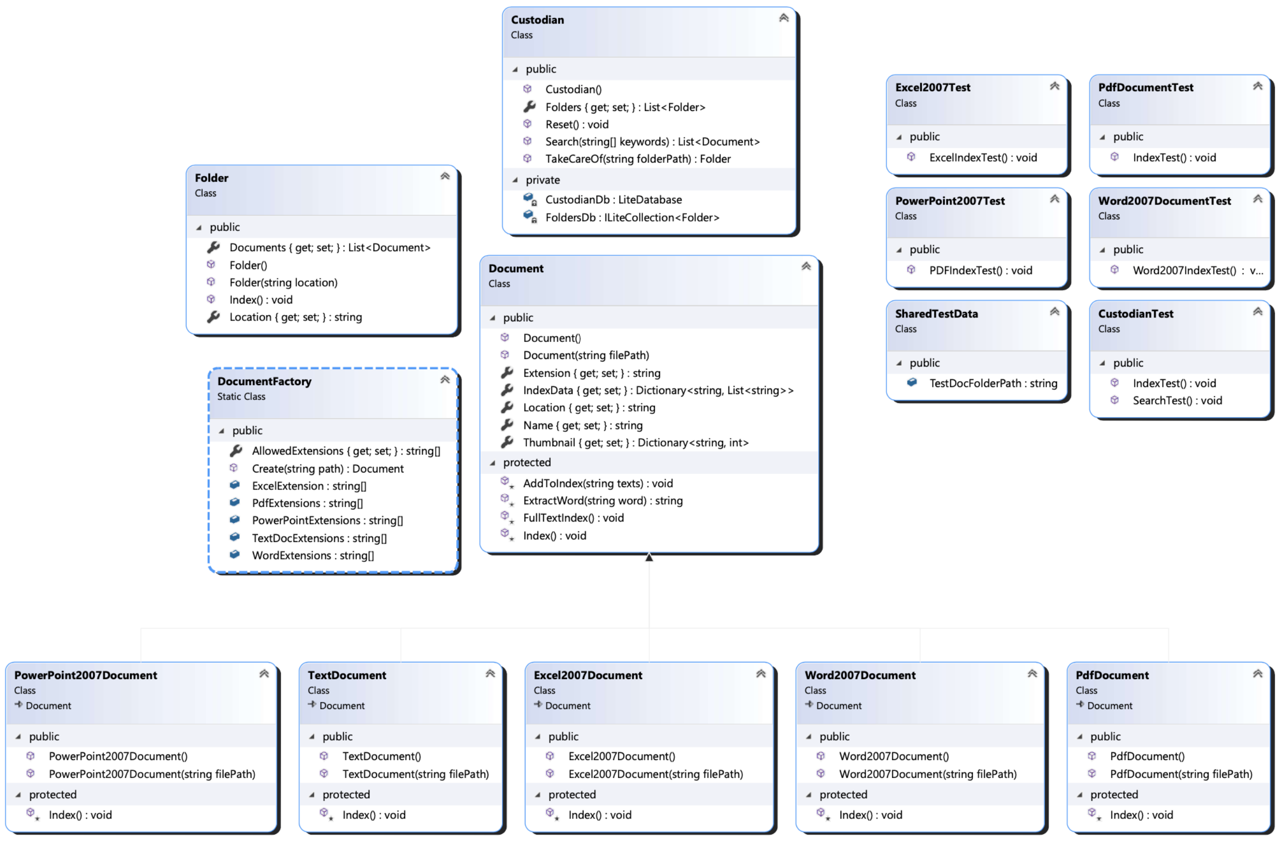
At first I planed to build both GUI (Graphic User Interface) and API (Application Programming Interface), which later turns out to be not feasible enough given the complexity of implementing GUI document previewer and the time constrain for this project. That said, I decided to shift my focus onto API alone.

Tika came into the picture, as I was going through DocFetcher's source code try to understand the implementations for its powerful features. Given the fact that there're no good alternatives to Tika in C# ecosystem, then it occurs to me that i should implement a similar API for text-extraction or content-analysis in C#.

Requirements Analysis

[sth. about requirement analysis in general]

API Design


Custodian
Class
public
Custodian()
Folders { get; set; } : List<Folder>
Reset() : void
Search(string[] keywords) : List<Document>
TakeCareOf(string folderPath) : Folder
private
CustodianDb : LiteDatabase
FoldersDb : ILiteCollection<Folder>
Folder
Class
public
Documents { get; set; } : List<Document>
Folder()
Folder(string location)
Index() : void
Location { get; set; } : string
DocumentFactory
Static Class
public
AllowedExtensions { get; set; } : string[]
Create(string path) : Document
ExcelExtension : string[]
PdfExtensions : string[]
PowerPointExtensions : string[]
TextDocExtensions : string[]
WordExtensions : string[]
Document
Class
public
Document()
Document(string filePath)
Extension { get; set; } : string
IndexData { get; set; } : Dictionary<string, List<string>>
Location { get; set; } : string
Name { get; set; } : string
Thumbnail { get; set; } : Dictionary<string, int>
protected
AddToIndex(string texts) : void
ExtractWord(string word) : string
FullTextIndex() : void
Index() : void
Excel2007Test
Class
public
ExcelIndexTest() : void
PdfDocumentTest
Class
public
IndexTest() : void
PowerPoint2007Test
Class
public
PDFIndexTest() : void
Word2007DocumentTest
Class
public
Word2007IndexTest() : v...
SharedTestData
Class
public
TestDocFolderPath : string
CustodianTest
Class
public
IndexTest() : void
SearchTest() : void
PowerPoint2007Document
Class
Document
public
PowerPoint2007Document()
PowerPoint2007Document(string filePath)
protected
Index() : void
TextDocument
Class
Document
public
TextDocument()
TextDocument(string filePath)
protected
Index() : void
Excel2007Document
Class
Document
public
Excel2007Document()
Excel2007Document(string filePath)
protected
Index() : void
Word2007Document
Class
Document
public
Word2007Document()
Word2007Document(string filePath)
protected
Index() : void
PdfDocument
Class
Document
public
PdfDocument()
PdfDocument(string filePath)
protected
Index() : void

[API WorkFlow]

Functional Requirements and Progress

Icon	☐	✓	🚧
Meaning	Planed	Done	WIP

- API
  - ✓ 64-bit multi-platform support.
  - ✓ Indexing.
  - ✓ Searching.
  - ☐ Semantic Search. !!!
  - ✓ Web API + API Documentation.
- File type support

- [x] Microsoft Office 2007+ documents support.
  - [x] `.docx`, `.docm` | Word 2007+ documents.
  - [x] `.pptx`, `.pptm` | PowerPoint 2007+ documents. ⚠
  - [x] `.xlsx`, `.xlsm` | Excel 2007+ documents. ⚠
- [ ] Plain-text documents.
  - [x] `.txt` | Text documents.
  - [ ] `.md` | Markdown documents. 🚧
  - [ ] `.rtf` | Rich text format documents.
- [ ] `.pdf` | PDF documents.
  - [x] Text-based PDF.
  - [ ] Image-only PDF. 🚧
- [ ] `.epub`, `.mobi`

- Testing

Icon	Meaning
✅	Fully Tested
✓	Partially Tested (Basic functionalities)

File Type	Testing Status
`.txt`	✅
`.docx`, `.docm`	✅
`.pptx`, `.pptm`	✓
`.xlsx`, `.xlsm`	✓
`.pdf`	✓

Implementation

Third-party Libraries

This project also utilizes several third-party libraries to implement certain functionalities.

LiteDB

LiteDB is a embedded No-SQL single-file Database. It's used in this project as an object storage for indexed data.

Optical Character Recognition

OCR Library

Open XML SDK

Open-XML SDK is an SDK that provides tools for working with Office (2007+) Word, Excel, and PowerPoint documents developed by OfficeDev Team at Microsoft.

Office Open XML (aka. OpenXML or OOXML) is an XML-based format for office documents, including word processing documents, spreadsheets, presentations, as well as charts, diagrams, shapes, and other graphical material. Since Office 2007 the file formats and structure of all office documents has changed. Instead of older binary formats (i.e. `.doc`, `.xls`, and `.ppt`), they have adopted Open XML to be the default format of all Microsoft Office documents (`.docx`, `.xlsx` and `.pptx`) (Office Open XML, 2012).

OpenXML files can be decompressed into many `.xml` files by compression software or tools.

Specific Implementations

Document Indexing

Word 2007

The structure of composing `.xml` files for `.docx` documents is similar to the picture (see Appendix c.).

The main document story of the simplest WordprocessingML (`.docx`) document consists of the following XML elements (Microsoft Docs, 2017):

Element	Description
document	The root element for a WordprocessingML's main document part, which defines the main document story.
body	The container for the collection of block-level structures that comprise the main story.
p	A paragraph.
r	A run.
t	A range of text.

Following is the code to extract text from Word documents.

```
protected override void Index()
{
    var startTime = DateTime.Now;
    Console.Write($"Indexing {Name}");

    #region Index Word Document
    // ReadFiles
    using var doc = WordprocessingDocument.Open(path: Location, isEditable: fals
    var body = doc.MainDocumentPart.Document.Body;
    // Get all <w:p> elements.
    using var paragraphParts = body.Descendants<Paragraph>().GetEnumerator();
    while (paragraphParts.MoveNext())
    {
        var currentParagraph = paragraphParts.Current;
        // Find all <w:t> elements inside each <w:p> element.
        var textParts = currentParagraph.Descendants<Text>().AsQueryable();
        var paragraphTemp = from text in textParts
                            where text.Text != ""
                            // Last char is number, possibly page number in Tab
                            select char.IsNumber(text.Text, text.Text.Length - 
        var paragraph = string.Concat(paragraphTemp);

        if (new[] { "" }.Contains(paragraph)) continue;
        if (paragraph == null) continue;

        this.AddToIndex(texts: paragraph);
    }
    #endregion

    Console.Write(
        $" >==> {Thumbnail.Count} unique words. {(DateTime.Now - startTime).Tota
}
```

Excel 2007

```
protected override void Index()
{
    var startTime = DateTime.Now;
    Console.Write($"Indexing {Name}");

    #region Excel
    var xlsx = SpreadsheetDocument.Open(path: Location, isEditable: false);
    var workSheets = xlsx.WorkbookPart.WorksheetParts.GetEnumerator();

    # region Get sharedStringTable
    IQueryable<string> sharedStringTableList = null;
    using (var parts = xlsx.RootPart.Parts.GetEnumerator())
    {
        while (parts.MoveNext())
```

```
            {
                var b = parts.Current.OpenXmlPart;
                if (b.GetType() == typeof(SharedStringTablePart))
                {
                    var part = (SharedStringTablePart)parts.Current.OpenXmlPart;
                    sharedStringTableList = from item in part.SharedStringTable.Eler
                                            select item.InnerText;
                }
            }
        }
        #endregion

        while (workSheets.MoveNext())
        {
            var sheet = workSheets.Current;
            using var sheetData = sheet.Worksheet.Elements<SheetData>().GetEnumerato
            while (sheetData.MoveNext())
            {
                using var rows = sheetData.Current.Elements<Row>().GetEnumerator();
                while (rows.MoveNext())
                {
                    var row = rows.Current;
                    using var cells = row.Elements<Cell>().GetEnumerator();
                    while (cells.MoveNext())
                    {
                        var cell = cells.Current;
                        var text = "";
                        // TODO: Potential issue with the cell type
                        // FIXME: The value the cells with type `Date`
                        if (cell.DataType == null)
                            text = cell.CellValue.InnerText;
                        else if (cell.DataType == CellValues.SharedString)
                            text = sharedStringTableList.ElementAt(int.Parse(cell.Ce
                        else
                            text = cell.CellValue.InnerText;

                        this.AddToIndex(text);
                    }
                }
            }
        }
        #endregion

        Console.Write($" >==> {Thumbnail.Count} unique words. {(DateTime.Now - start
    }
```

PowerPoint 2007

```
protected override void Index()
{
    var startTime = DateTime.Now;
    Console.Write($"Indexing {Name}");
```

```
        # region PowerPoint
        var ppt = PresentationDocument.Open(path: Location, isEditable: false);
        using var slides = ppt.PresentationPart.SlideParts.GetEnumerator();

        while (slides.MoveNext())
        {
            var slide = slides.Current;
            using var text = slide.Slide.Descendants<TextBody>().GetEnumerator();
            while (text.MoveNext())
            {
                this.AddToIndex(texts: text.Current.InnerText);
            }
        }
        #endregion

        Console.Write($" >==> {Thumbnail.Count} unique words. {(DateTime.Now - start
    }
```

Tools

C Sharp

.NET Core

ASP.NET Core

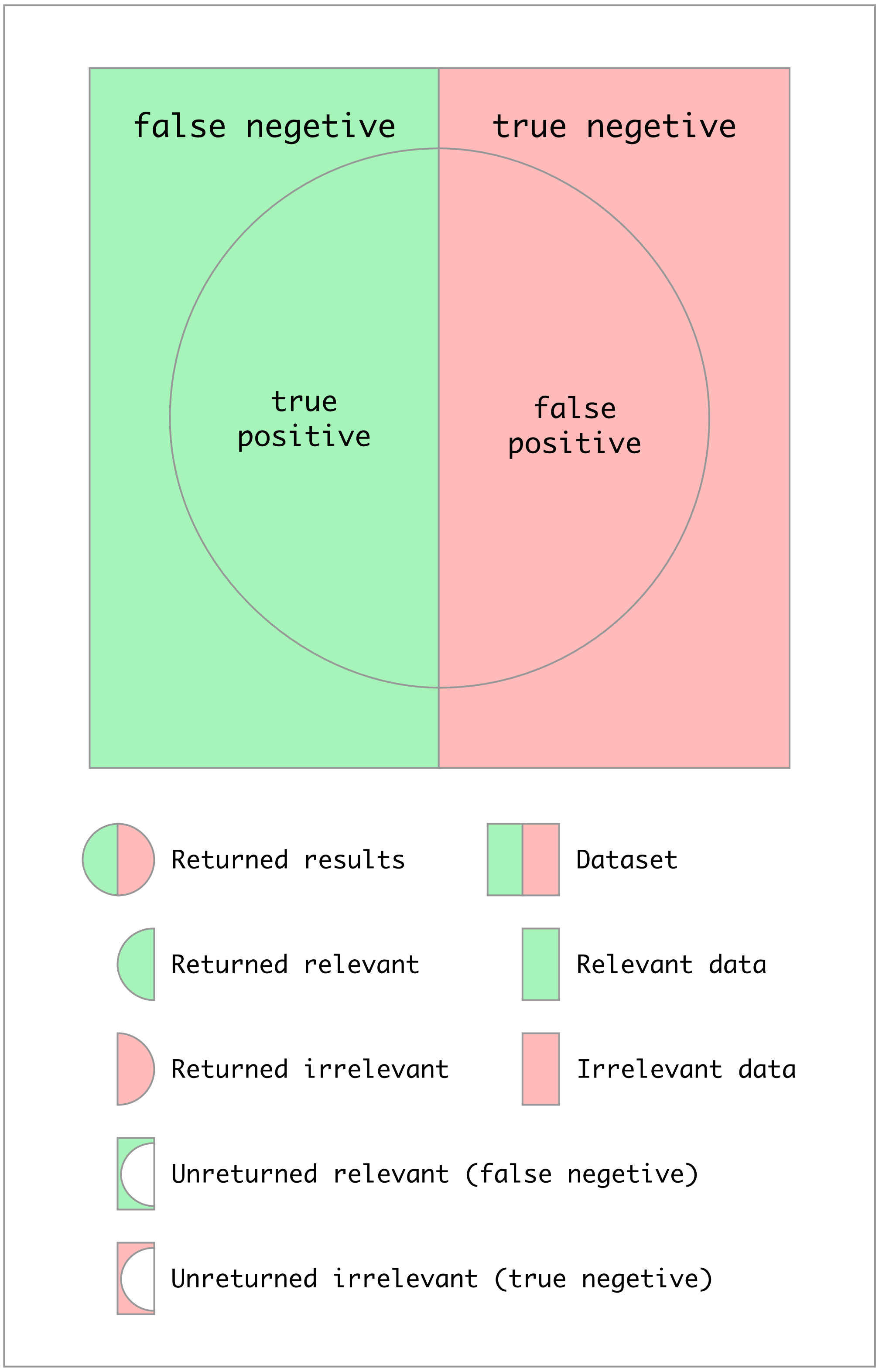
Visual Studio Code

References

- Erickson, 2008. sql - What is Full Text Search vs LIKE - Stack Overflow. Accessed on July 1st, 2020.
- Makhoul, J., Kubala, F., Schwartz, R. and Weischedel, R., 1999, February. Performance measures for information extraction. In Proceedings of DARPA broadcast news workshop (pp. 249-252).
- Melton, J. Buxton, S., 2006. Chapter 13 - What's Missing? - Querying XML | ScienceDirect. Accessed on July 1st, 2020.
- Microsoft Docs, 2018. Full-Text Search - SQL Server | Microsoft Docs. Accessed on July 1st, 2020.
- Reiner, K. Chichao, C. Farzin, M. Ravi, K.,2006. Searching with context | ACM Digital Library. Accessed on July 1st, 2020.
- Tekli, J., Chbeir, R., Traina, A., Traina, C Jr., Yetongnon, K., Ibanez, C., Assad, M., Kallas, C., 2018. Full-fledged semantic indexing and querying model designed for seamless integration in legacy RDBMS. Accessed on July 2nd, 2020.
- Office Open XML, 2012. Office Open XML - What is OOXML? Accessed on 5th August, 2020.
- Microsoft Docs, 2017. Structure of a WordprocessingML document (Open XML SDK) | Microsoft Docs. Accessed on 5th August, 2020
- Apache Tika, 2020. Apache Tika - Apache Tika. Accessed on 7th Aug, 2020.
- Apache Lucene, 2020 Apache Lucene - Welcome to Apache Lucene. Accessed on 7th Aug, 2020.
- DocFetcher, 2020 DocFetcher - Fast Document Search. Accessed on 7th Aug, 2020.

Appendix

a. Precision & Recall | Back to content



b. Gantt Chart | Back to content

c. Structure of decompressed `.docx` file | Back to content