1406Z Course Project Report
Web Crawler and Search Engine GUI

Seyedsajad Hosseini

December 2022

How To Run the Code

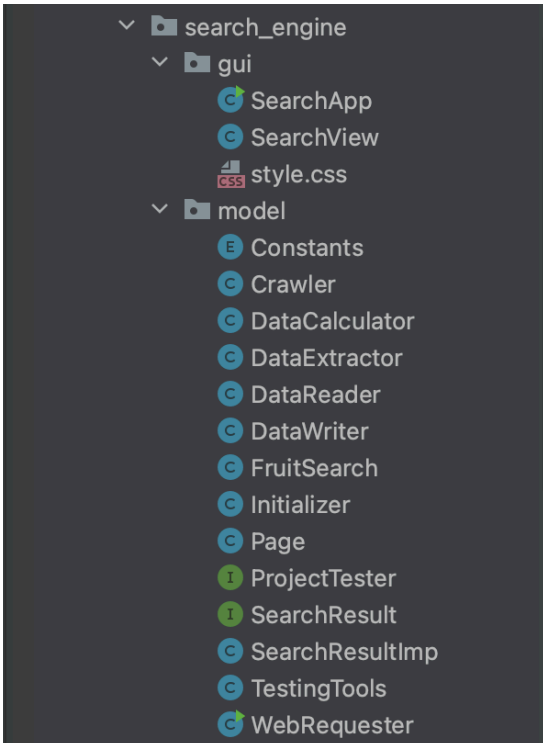
- Crawler and Fruits Databases
    - All the files necessary to run the crawler are in the package named "*search_engine*". To make the execution of tester files for each database more straightforward, six packages have been created for *tinyftuits* and *fruits1* to *fruits5* databases. A copy of *search_engine* sub-package and all necessary test files for each database have been placed in these packages.
    - To run a test file for a specific database, just open the package created for that database and find the specific test file.
    - All the files storing the crawled data will be saved in a folder named "*data*" inside each package.
    - All the failed/passed files produced by the test process are also stored inside each package (To change the location of these files, two lines of each test file that create a *PrintWriter* have been modified).
- GUI
    - To run the GUI, one must first run the crawler so that the "data" folder is created. Then run the *SearchApp* class which can be found using the following pattern:

fruitsX/tinyfruits ⇒ search_engine ⇒ gui ⇒ SearchApp

Project Structure

Introduction

The classes created for this project are placed into two sub-packages of *search_engine* package: *model* sub-package that contains all the crawl and search-related classes necessary to run the test files, and *gui* sub-package that contains classes that create and control the user interface (picture 1).



Picture 1. Hierarchy of classes in the program

The following table gives a brief description of different classes developed for different parts of the program.

Table 1. Classes Used in the Program

Which Part of GUI	Class Name	Responsibility
Model	Page	creates objects that store each webpage data
	Crawler	implements ProjectTester
	Initializer	A helper class that deletes last crawl data and creates the *data* folder
	DataExtractor	A helper class that extracts webpage data (title, links, words)
	DataCalculator	A helper class that performs calculations (e.g. page rank) on raw data
	DataWriter	A helper class that writes data to files
	DataReader	A helper class that reads data from files
	FruitSearch	A helper class that performs the search
	SearchResultImp	A class that implements SearchResult interface
	Constants	stores constant names, numbers, and file addresses
Controller	SearchApp	executes the GUI
View	SearchView	creates JavaFX controls
	style.css	contains stylings for GUI objects

Page Class

The data crawled and calculated for each webpage is stored in a *Page* object. The object's attributes are introduced in table 2 (the names are descriptive of each attribute's role).

As the exact number of incoming and outgoing links are not known beforehand, a *List<String>* is a better choice than *String[]* to store these data. In addition, as each word (fruit) has a different *tf* and *tf-idf* value for every Page object, a *Map* is necessary to save store such key-value pairs.

Table 2. Page class attributes

Attribute Name	Type	Crawled/Calculated by*
link	String	crawl()
title	String	DataExtractor.extractTitle()
pageRank	double	DataCalculator.calculatePageRank()
linksOut	List<String>	DataExtractor.extractLinks()
linksIn	List<String>	DataCalculator.findIncomingLinks()
tf	Map<String, Double>	DataCalculator.calculateTfInUrl()
tfIdf	Map<String, Double>	DataCalculator.calculateTfIdf()

* method input is not shown

During the crawl process, one *Page* object is created for each visited page. At the end of the crawl, each attribute of the created *Page* object is written to a text file. To improve file reading efficiency during the search, each key-value pair in *tf* and *tfIdf* maps is written to a single text file. During the search process, these written files are read and used in the calculations.

Crawler Class and its Helper Classes

This class implements *ProjectTester* interface and is responsible for crawling, calculating, storing, and retrieving information. Except *crawl()* methods, other methods don't have a body except to return the values returned by static methods of helper classes. These classes are designed to read and extract data from HTML, calculate the required variables, and write them to the files. Table 3 shows the *Crawler* class attributes.

wordsCountInPage is a map in which key is a word (fruit) and value is the number of occurrences of that word in the webpage. *wordsCountTotal* is similar to the *wordsCountInPage* except it adds up the number of occurrences of each word in all the visited webpages. *wordsRelFreqInPage* is a map in which key is a word (fruit) and value is the number of occurrences of that word divided by all the words counted in the crawl.

Table 3. Crawler class attributes

Attribute Name	Type	Crawled/Calculated by*
pages	Map<String, Page>	crawl() DataExtractor.extractTitle() DataExtractor.extractLinks() DataExtractor.calculateTfInUrl() DataCalculator.calculatePageRank() DataCalculator.findIncomingLinks() DataCalculator.calculateTfIdf()
queueUrls	Queue<String>	crawl()
processedUrls	List<String>	crawl()
wordsCountInPage	Map<String, Integer>	DataCalculator.calculateWordCount()
wordsCountTotal	Map<String, Integer>	DataCalculator.updateWordsCount()
wordsRelFreqInPage	Map<String, Double>	DataCalculator.calculateTfInUrl()

* method input is not shown

The *Crawler* class overrides the following methods form the *ProjectTester* interface.

- *initialize()*

  This method calls *Initializer.initialize()* static method to delete the files and folders created by the last crawl and create a new base folder. To traverse all files and folders, the built-in method *Files.walkFileTree()* is used. This method needs a *SimpleFileVisitor* object which itself implements *FileVisitor* interface and has methods for visiting files and directories. These methods can be designed in a way that visited files and directories are deleted (a directory is deleted only after it's become empty).

  After deletion of all files and folders, a new base folder called "*data*" is created to store data from the current crawl.

- *crawl()*

  crawl() receives a URL as input and starts the crawl process by reading HTML of that webpage, extracting links to other webpages, words (fruits), and title

of the webpage. The extracted webpage URLs fuel further crawls until all webpages are read. The extracted words are used for calculation of term frequencies (tf), inverse document frequencies (idf), page ranks, and tf-idf values. At the end, the extracted and calculated data are written to text files. To do so, the helper classes shown in table 4 are utilized.

Table 4. Crawler class helper functions

Helper Class	Method Name*	Task
DataExtractor	extractLinks()	finds absolute webpage addresses that the current webpage links to from its html returns an ArrayList<String> of found addresses
	extractWords()	finds the words (fruits) between / tags in the webpage html returns an String[] of found words
	extractTitle()	finds and returns the title of the webpage from its html (stored between <title> / </title> tags)
DataCalculator	calculateWordCount()	counts number of occurrences of each word in the webpage returns a map where key: word in the webpage, value: number of occurrences of the word
	updateWordsCount()	updates a map that counts occurrences of each word read in the webpages returns a map where key: a word read in one or more of the visited webpages, value: updated number of webpages with the word
	calculateTfInUrl()	calculates tf value for each word in the current webpage returns a map where key: word in the webpage, value: word frequency (tf) in the webpage
	findIncomingLinks()	finds incoming links to a webpage based on other webpages' outgoing links adds found incoming links to the webpage Page object's linksIn attribute
	calculateTfIdf()	calculates tf-idf value for each page and saves it in the page tfIdf attribute
	calculatePageRank()	calculates each webpage's page rank and saves it in the pageRank attribute of that webpage

	matrixMult()	calculates matrix multiplication of an array and a matrix, used in calculatePageRank()
DataWriter	writeIdfData()	calculates and writes idf data of all the words in the visited webpages to a directory called "IDF" in the first layer of the directory hierarchy. "IDF" stores one text file for each word.
	writePageData()	writes attributes of each webpage (Page object) to a file in a directory named after that page (e.g. N-1.html). Each directory contains: (1) a "TF" directory which stores tf values for each word in that webpage in a text file (e.g. /N-1.html/TF/orange.txt is the address of text file that stores tf value of orange) (2) a "TF_IDF" directory which stores tf_idf values for each word in that webpage in a text file (e.g. /N-1.html/TF_IDF/orange.txt is the address of text file that stores tf_idf value of orange) (3) an "incoming links.txt" file storing the addresses of webpages with a link to the current webpage (4) an "outgoing links.txt" file storing the addresses of webpages that the current webpage has a link to (5) a "page_rank.txt" file storing the page rank calculated for the current webpage (6) a "title.txt" file storing the webpage title
	writeUrls()	writes all the visited URL addresses to the file "URLs.txt" in the first layer of directory hierarchy.

* method input is not shown

- *getOutgoingLinks(), getIncomingLinks(), getPageRank(), getIDF(), getTF(), getTFIDF()*

  The above methods return the data saved in the crawl process. To make the program modular, a helper class named *DataReader* with static methods has been created. The overridden methods in the *Crawler* perform no further processing on the information returned by *DataReader* methods.

To read the files stored in various folders, first the specific folder should be specified based on the input URL. This can be achieved using the helper method *findUrlFolder()*. Then, based on the specific parameter requested, one of two overloaded *readFile()* methods or *readTitleFiles()* method is used to read the data. *readFile()* methods return a number of type double or a List<String> based on their input, while *readTitleFiles()* method returns a String[].

- *search()*

  This method runs the *search()* method in the helper class *FruitSearch*, which finds the necessary data in the memory and calculates similarity index for each crawled URL. To implement the *SearchResult* interface, a class named *SearchResultImp* has been created which also implements *Comparable* to help sorting the results based on, first, their similarity index and, second, their title alphabetic order.

OOP Incorporation

The code is mainly composed of methods that perform calculations or write data to files or read them to be presented to the user. Therefore, while some of the principles of OOP have been incorporated into the project code, there seems to be no need for a hierarchy of classes where inheritance aspect of OOP can be applied.

The written code is, however, modular: separate parts of the program can perform their tasks and connect with each other without the knowledge of internal process of other parts. Helper methods are all in various separate classes and perform their task independently. For instance, the code written for the GUI sends the query to the search method and receives the results. Therefore, the search method or GUI can be changed completely with no effect on each other.

In addition, application of abstraction can be observed in different parts of the code. During the design process of the program, the crawling mechanism was divided into three separate parts: extracting data from webpage HTML, calculating the required variables, and writing them into files. Each part performs its task abstracted from other parts. For instance, after the data is extracted from the webpage HTML the *crawl()* method just calls write methods from *DataWriter* class and tell them to write a map of Page objects, it is ignorant if each Page object is being written into one file or as separate attributes into several files. This is a helpful notion when extensibility of the program is considered. One can add more code to the *DataExtractor*, so that it extracts more information from the webpage. The write and read methods can continue performing their task after addition of new attributes to the Page object with minimal adjustment to the code.

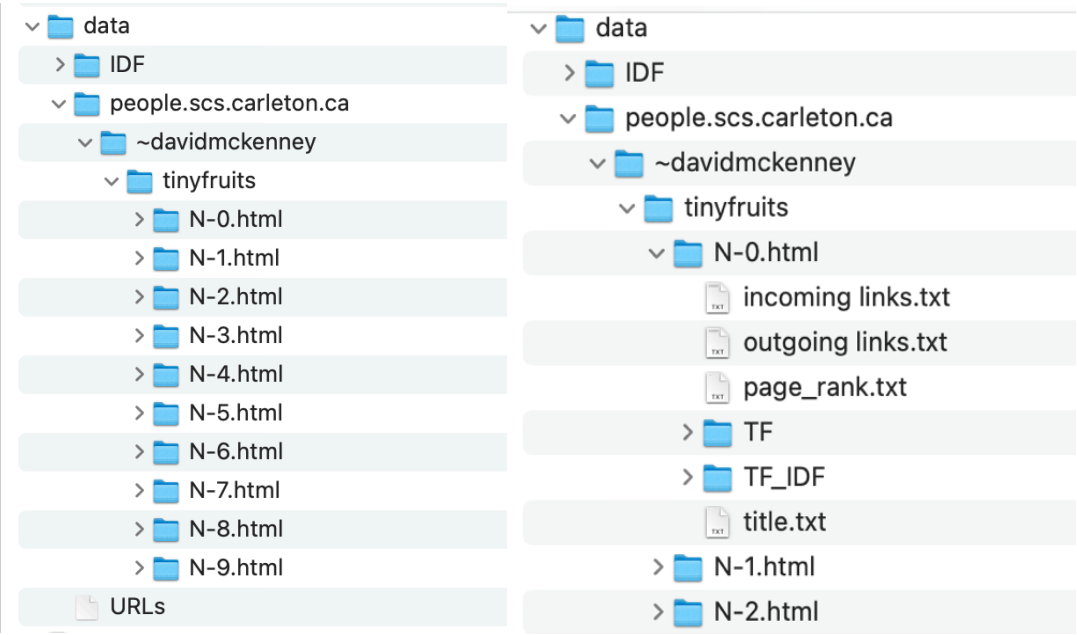
Using overloaded methods is another example of abstraction implemented in this project. These methods have been coded in a way that satisfy the file reading needs of *get…()* methods with less amount of code. This, in addition to *findUrlFolder()* method, improves code readability and reusability.

Encapsulation is another principle of OOP being incorporated into the program. Attributes of the Page object are private and can only be accessed through getter and, for some attributes, setter methods. While some of the benefits of OOP implementation are evident in the final code, other advantages of taking the OOP path, such as easier debugging and modification, were put to use during the development of the program.

Data Storage Structure

While for the same project written in Python, several JSON files were used to store dictionaries of data, as there's no built-in JSON parsing library in Java, another

approach had to be taken. Built upon the experience of developing different directory hierarchies for the Python project, the one shown below was chosen for this program.



Picture 2. Data storage structure: webpage folders (left), files and folders in a webpage folder (right)

The data for each crawled webpage is stored in a folder whose path is determined by the webpage's URL. For the fruits databases, as all URLs differ only in the last part, all the N-x folders can be moved to the base folder ("data"). However, the current structure guarantees utilization of the program for more complex situations where webpages are from various domains and subdirectories.

Discussion on Relation Between Project Structure and Data Storage

For each webpage, data is stored in several text files. *incoming links.txt*, *outgoing links.txt*, and *page_rank.txt* contain data requested by *getIncomingLinks(), getOutgoingLinks(),* and *getPageRank()*, respectively. In addition, TF and TF_IDF directories each contain one file per word (fruit) in the webpage. Therefore, when *getTF()* and *getTFIDF()* methods access a specific file based on the input URL and word, there is no need to read TF and TF-IDF values of other words. If, for instance,

all the data was saved in one dictionary, in a text file context, the code had to search the dictionary for the specified URL.

In conclusion, the data storage structure is based on the requirements of the *Crawler* class. Whenever *get* methods are given a URL (and a word, in case of TF and TF-IDF) as the input and exectued

- as the names of folders and URLs match, the code knows where to search for the correct data,
- as there is no need for processing the file content, the specific file is read, and its content is simply returned.

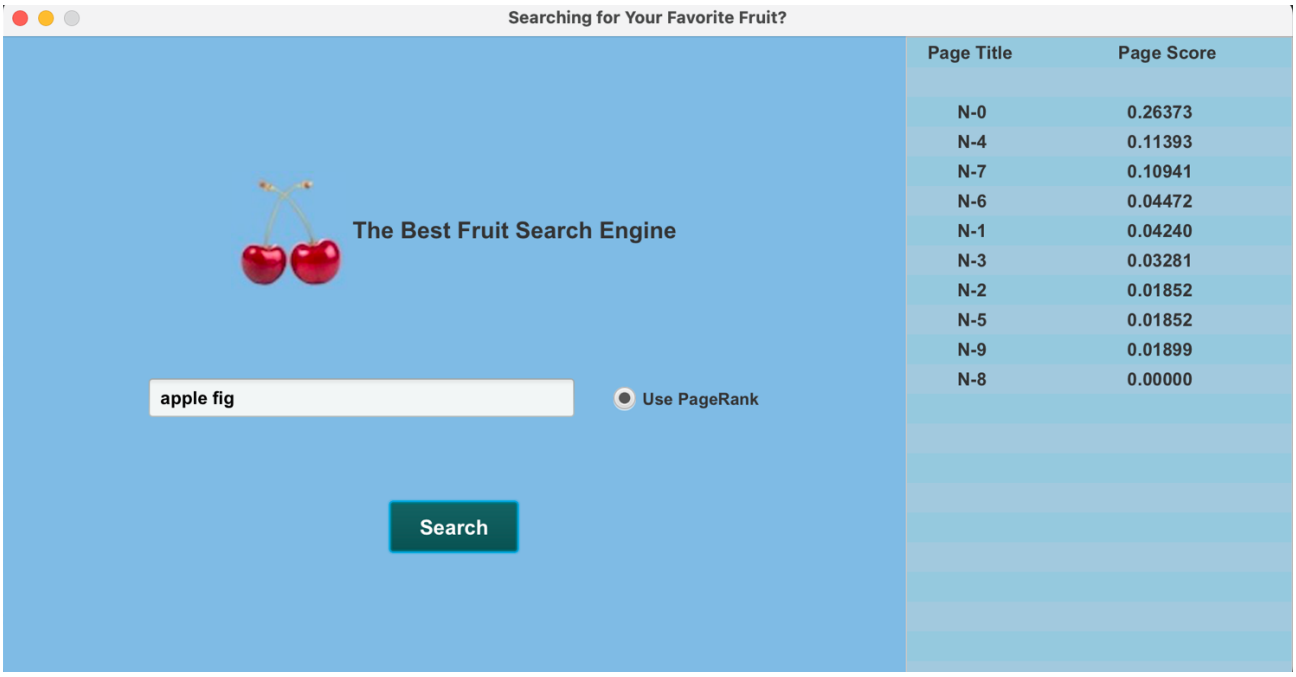
One alternative to the current structure is making the Page object Serializable and saving Page objects to files. This approach was examined but decided not to be followed. The main benefits and shortcomings of such approach for the current project are shown in the following table.

Table 5. Benefits and limitations of saving data as Page objects

Advantages	Disadvantages
Less storage space	No readable output
	Must load all Page attributes to read a specific one
Less write/read code	Needs a Page object in the get methods of the ProjectTester interface

GUI Design

Picture 3 shows the user interface for the search section of the program. It's been designed based on the project specifications document and has a Textfield for entering fruit names, a RadioButton for selecting/deselecting use of page rank in the calculations, a search Button, and a ListView showing top 10 results.



Picture 3. Search program user interface

Addition of a second TextField and Button for crawling new webpages was considered. Because the crawling process for *fruits1* to *fruits5* databases take more than 2 minutes and in the meantime the user cannot use the interface, it may seem to the user that the program is irresponsive or has crashed. Therefore, it was decided to not add crawl capability to the GUI. To use the interface for a new set of webpages, one must first crawl those webpages using the *Crawler* class and then run the GUI.

A Summary of Design Approaches

- Hierarchy of directories based on webpage's URL

  File structure design affects all methods that need to write or read data to perform their task. As described in a previous section, saving the data in text files results in a hierarchy of directories whose naming is based on the visited URLs. Each file in the file structure corresponds to a *get* method in the code, making it fast and straightforward to read and write data.
- Use of helper and overloaded methods

When used instead of one longer complex method, helper methods improve readability of the program. Each helper method with a specific function can be used in several other methods. For example, in this project, *findUrlFolder()* is a helper method responsible for creating the path of a folder that contains the saved data for a webpage. Application of this method in several *get* methods is an example of code reusability.

- Use of Java lambda expressions and streams

  Lambda expressions and streams result in a more concise and readable code. Although not improving efficiency, they have become more common especially for smaller code blocks. These relatively new features of Java were used both to improve code readability and as an exercise to learn new programming tools.

- Addition of a class containing shared names, numbers, and file addresses

  To reduce typing errors, improve code reusability and maintainability, all file names and constants are moved to a separate class called *Constants*.

- Addition of a css file

  To improve readability of the *SearchView* class, which is responsible for creating GUI controls, a stylesheet file containing code for font and color design of the user interface is created.

- runtime efficiency

  As the focus of this project is not optimizing the crawl-search process, less time was spent on measuring runtime and tuning the code performance. However, as the calculation methods are Java versions of the optimized Python code written for the same project, they can be considered highly efficient. The crawling-testing process for *fruits1* to *fruits5* databases take approximately 2.5 minutes on a MacBook Air M1 laptop.

An Explanation about Failed Search Cases

Among all test cases provided by the test files, only three fail[1] (table 6). A comparison between the results obtained from the codes written in Python and Java reveals that the exact same results are produced by both versions. While in the Python version no failed cases is found, in the Java version three failed cases happen due to rounding errors.

Table 6. Failed search cases

Database / Test Number	My Results	Expected Results
Fruits3 Test #565	1. N-6 = 0.015729262839602597 2. N-5 = 0.012097493274448924 3. N-0 = 0.011491238557678217 4. N-25 = 0.009964201626687466 5. N-10 = 0.008119138622370778 6. N-4 = 0.007917906721324593 7. N-2 = 0.00697528289300795 8. N-241 = 0.00594602151234714 9. N-3 = 0.0056675160080817385 10. N-64 = 0.005667890751216069	1. N-6 = 0.015723031366456176 2. N-0 = 0.011501238625641974 3. N-5 = 0.012099894535426335 4. N-25 = 0.00996849532855458 5. N-10 = 0.008118972693874545 6. N-4 = 0.007916513286929803 7. N-2 = 0.006972590806293313 8. N-241 = 0.005940792255330082 9. N-3 = 0.00566413383882965 10. N-64 = 0.005662950281767862
Fruits4 Test #529	1. N-12 = 0.01780135795176204 2. N-3 = 0.016686020791939586 3. N-0 = 0.015319225209860208 4. N-1 = 0.01490695527881243 5. N-4 = 0.00805635178757847 6. N-6 = 0.00849715431835548 7. N-15 = 0.006912470878577976 8. N-8 = 0.007242914353079262 9. N-14 = 0.005707226098757778 10. N-32 = 0.00591936272893824	1. N-12 = 0.01781937965888868 2. N-3 = 0.016690105574155144 3. N-0 = 0.015333582037452591 4. N-1 = 0.014914846556925345 5. N-6 = 0.008502665831652496 6. N-4 = 0.008059417696651522 7. N-15 = 0.006911132574330005 8. N-8 = 0.007241349187803715 9. N-14 = 0.005710820960302381 10. N-32 = 0.005917561159384446
Fruits4 Test #559	1. N-12 = 0.01780135795176204 2. N-3 = 0.016686020791939586 3. N-0 = 0.015319225209860208 4. N-1 = 0.01490695527881243	1. N-12 = 0.01781937965888868 2. N-3 = 0.016690105574155144 3. N-0 = 0.015333582037452591 4. N-1 = 0.014914846556925345

[1] Tests #529 and #559 in Fruits4 database are the same. In both tests the query is “tomato cherry fig kiwi fig” with boost=true. Therefore, as can be seen in table 6, the expected results exactly match.

	5. N-4 = 0.00805635178757847 6. N-6 = 0.00849715431835548 7. N-15 = 0.006912470878577976 8. N-8 = 0.007242914353079262 9. N-14 = 0.005707226098757778 10. N-32 = 0.00591936272893824	5. N-6 = 0.008502665831652496 6. N-4 = 0.008059417696651522 7. N-15 = 0.006911132574330005 8. N-8 = 0.007241349187803715 9. N-14 = 0.005710820960302381 10. N-32 = 0.005917561159384446

For the failed test of Fruits3 database, the positions of N-0 and N-5 should be swapped. The reason for this mistake is that the calculated score of N-0 (rounded to 5 decimals) is 0.01149, while the expected result is 0.01150. When rounded to 3 decimal points, the calculated and expected results equal 0.011 and 0.012, respectively. Therefore, the code positions N-0 after N-5, which has a rounded score of 0.012, while the expected result put N-0 above N-5, as they have the same rounded scores and N-0 is alphabetically in front of N-5.

The same issue is reproduced for Test #529 (and its equivalent Test #559). Here, N-6's score is calculated as 0.008497, whereas it's expected to be 0.008502. When rounded to three decimal points, the first number becomes 0.008 and the second 0.009. Therefore, a small difference of 0.000005 is converted into 0.001, changing the order of pages with score values close to X.XXX5.