Improving the Capabilities of FutureLens

Joshua M. Strange, Master's Candidate

Dr. Michael W. Berry, Major Professor

Department of Electrical Engineering and Computer Science
The University of Tennessee

Friday, June 29th, 2012
Knoxville, TN

Abstract

FutureLens is a powerful tool for data mining that aids in the discovery of knowledge by allowing the user to quickly see interesting patterns in data. However, FutureLens is currently limited in the amount of data that it can process at one time, and this limitation hinders FutureLens from reaching its full potential. This PILOT project expands the capabilities of FutureLens by increasing the amount of data that it can process, decreasing the amount of time that the user waits for data to process, and by adding features not included in the original implementation of FutureLens.

1. Motivation and Introduction

FutureLens was created by Gregory Shutt, and Dr. Andrey Puretskiy as a tool to allow the user to visualize data, and aid in the discovery of knowledge [1]. The original intent of FutureLens was to use any size of data, and see a visualization of the given data. However, FutureLens has an upper limit on the amount of data it can process at one time. The upper limit depends on the size of each document in the data set. If the documents have an average size of four kilobytes then FutureLens can process no more than seventy-five thousand documents, and FutureLens can only process five thousand documents when each document has an average size of forty-five kilobytes.

Thus, a goal of this PILOT is to find the cause of the data limitation in FutureLens, and reduce or eliminate it entirely from the application. While improving the size and amount of data that can be processed, the amount of time to process the data becomes important. Another, goal is to decrease the amount of time needed for FutureLens to process data. The final goal of the PILOT is to add features such as, an external stop list, and the ability for the user to trim the dictionary of terms based on a minimum, maximum, or range of frequencies.

To show the impact of changes made to FutureLens, data comparing the two versions was generated. All comparison data was generated using the default settings for FutureLens that I received, which means the Java virtual machine starts with one and a half gigabytes of memory, and can get up to two gigabytes of memory. The comparison data was created using two different data sets. The first data set contains abstracts from different psychological journals, and each document in the set has an average size of four kilobytes. The second set of data contains patent documents with an average document size of forty-five kilobytes.

The remainder of this report is arranged as follows: Section 2 describes the general obstacles faced before completing this PILOT project, Section 3 explains the additional functionality that has been added to FutureLens, Section 4 describes how the amount of data that FutureLens can process was increased, Section 5 discusses how the amount of time to process data was decreased, and Section 6 discusses future work that could be done to improve FutureLens.

2. General Obstacles

There were obstacles to resolve before work could begin on improving FutureLens. When I received FutureLens it would not work on my computer because FutureLens was created on a thirty-two bit system. This lead to the first obstacle to resolve, how to get FutureLens to work on a sixty-four bit system. The initial way to make FutureLens work on a sixty-four bit computer was to use a flag to make the Java virtual machine (JVM) operate in thirty-two bit mode. This answer did not seem like the best solution available. While attempting to build the source code of FutureLens, the libraries for the Standard Widget Toolkit (SWT) were identified as the problem. After upgrading to the sixty-four bit version of the SWT, FutureLens would work on my sixty-four bit computer without giving the JVM a flag to operate in thirty-two bit mode.

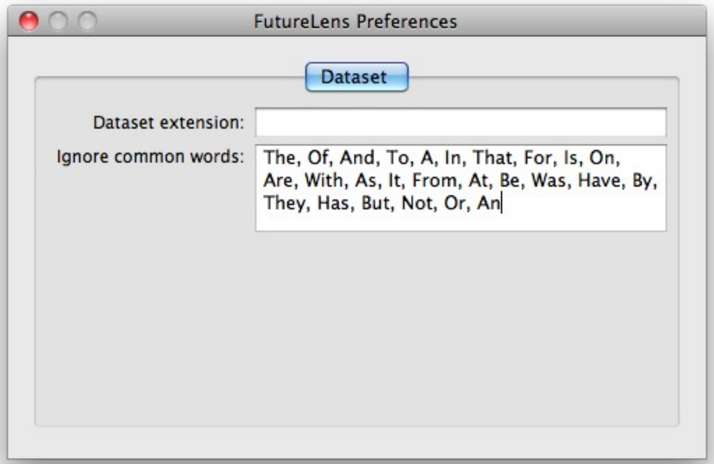
Being able to run FutureLens on my sixty-four bit computer exposed another problem that needed to be resolved before I could begin improving FutureLens, that problem was, the system created menu on my MacBook Pro no longer worked. Looking at the source code this was an issue because the preferences menu option allows the user to modify the list of words to ignore while FutureLens is processing data. The first step to resolve this issue was to find out why this option no longer worked. After, doing some research the cause of the problem was Apple changing the system API when going from OSX 10.5 to OSX 10.6. Armed with this knowledge the next step was to get the menu option working again. This involved changing how the menu bar widget was handling events from the system menu. To handle the events from the system menu an ArmListner, a type of event listener provided by SWT, was needed to allow FutureLens to know when options in the system menu were being used [2]. With the addition of the new event listener, the entire menu, in particular the preferences menu item, was working again.

3. Added Functionality

This PILOT added two features not in the original version of FutureLens. The first feature is an external stop list that the user can edit inside or outside of FutureLens. The second added feature allows for the user to create a custom dictionary based on a minimum, maximum, or range of term frequencies. This will allow the user to focus on terms that occur at a frequency that they choose without having to sort through pages of terms to find a given frequency.

3.1 External Stop List

A stop list is a list of terms that are to be ignored while processing documents in text mining. Originally FutureLens had a stop list implementation that could be accessed through the preferences menu option. It can be seen in Figure 1 shown below.

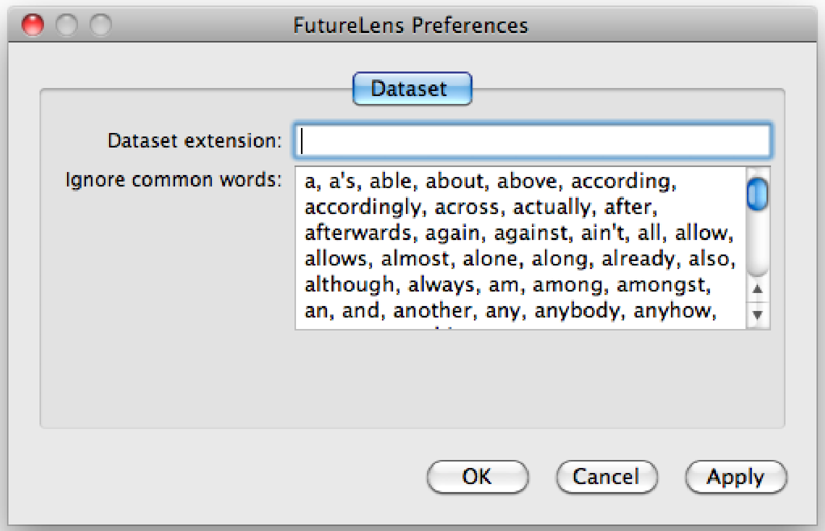


Dataset extension:

Ignore common words: The, Of, And, To, A, In, That, For, Is, On, Are, With, As, It, From, At, Be, Was, Have, By, They, Has, But, Not, Or, An

Figure 1: The original stop list in FutureLens.

The user could modify the terms on the list, but the changes were not guaranteed to be saved. Also, the importing or exporting of a stop list would require running FutureLens and then copying or pasting the stop list.To resolve these problems, an external stop list was implemented that would allow the user to modify the list at any time, and save changes that were made to the list while using FutureLens. The new stop list can be seen in Figure 2 below.



Dataset extension:

Ignore common words: a, a's, able, about, above, according, accordingly, across, actually, after, afterwards, again, against, ain't, all, allow, allows, almost, alone, along, already, also, although, always, am, among, amongst, an, and, another, any, anybody, anyhow,

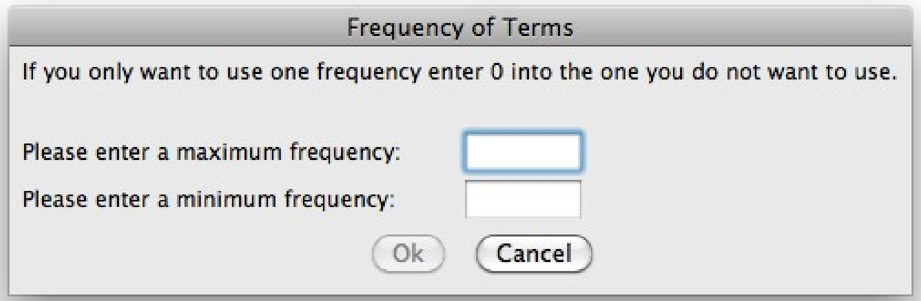
Figure 2: The upgraded stop list from FutureLens. The number of terms have increased and a scroll bar was added to let the user scroll through the words in the external stop list.

3.2.1 External Stop List Implementation

The external stop list is implemented as a standard text file with the name of Stoplist.txt or stoplist.txt. Also, a class named Stoplist contains all functions and private variables that are needed to allow for the use of an external stop list. If FutureLens can not find the stop list, then it will create a basic stop list for the user, in the correct location depending on what operating system the user is running. Next, depending on what the user is doing different functions of the Stoplist class are called. If the user has opened the preferences menu, then the Stoplist class is used to read in the text file, and format the text box into what is seen in Figure 2. While in the preferences menu, the user can add or delete terms from the shown list. When the user clicks the apply button, the shown stop list will be saved to the external text file. If the user has decided to load a data set, an ArrayList is populated with the contents of the external stop list before being passed to the parser.

3.3 Custom User Dictionary

In the original version of FutureLens, if the user wanted to find terms with a certain frequency they would have to go through pages of terms to find the desired frequency. This becomes very time consuming when there are millions of terms, and the frequency that is needed is hundred of pages away. However, with the added ability to create a custom dictionary, the user can easily find the frequency they need. To create a custom dictionary the user selects the create dictionary option in the tools menu. Once this option has been selected the following pop-up box will appear.


Frequency of Terms

If you only want to use one frequency enter 0 into the one you do not want to use.

Please enter a maximum frequency:

Please enter a minimum frequency:

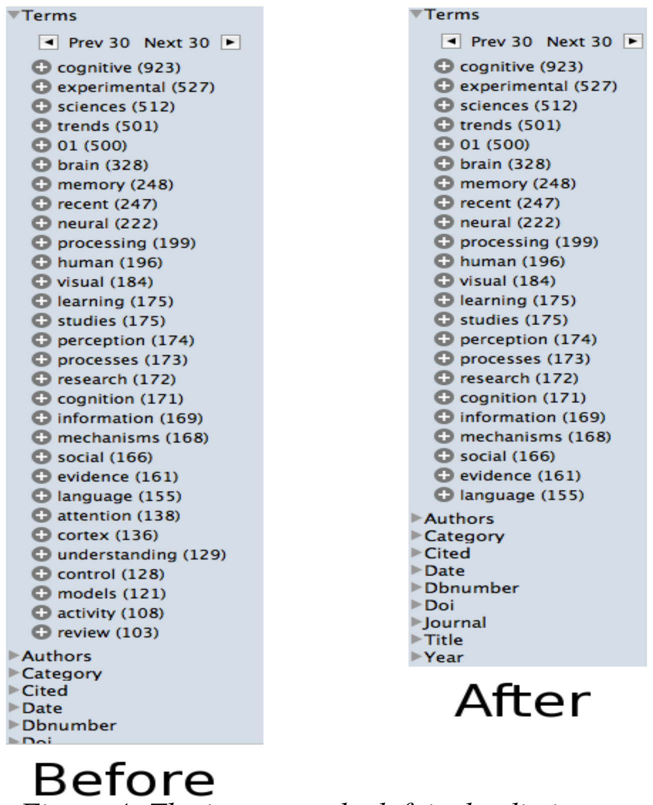
Figure 3: Box to enter range of frequencies.

The user can then make different types of dictionaries based on the values they enter. A dictionary were all terms have a frequency greater than or equal to a minimum is created by the user entering a value into the minimum frequency box, and then entering a zero into the maximum frequency box. To create a dictionary with terms that have a frequency that is less than or equal to a given frequency, the user enters a number for the maximum frequency, and then enters zero for the minimum frequency. For generating a dictionary with a range of frequencies the user enters the upper bound of the range as the maximum frequency, and the lower bound of the range as the minimum frequency. The created dictionary contains only terms with a frequency between the upper bound and lower bound inclusively.

3.3.1 Custom User Dictionary Implementation

The custom user dictionary is implemented as a simple sort of the parser generated dictionary. If the user wants to create a custom dictionary by selecting the create dictionary option in the tools menu, the box from Figure 3 is shown. The user input box has error checking to make sure the user enters valid input into both text boxes. This error checking is accomplished by using SWT's ModifyListener, which is attached to the text boxes. When the contents of one box change, an event is triggered that tries to convert the contents of the text box into an integer. If the conversion succeeds, the same procedure is tried on the other text box. While both text boxes contain valid input, the ok button is enabled. However, when one text box has valid input, and the other is blank or does not contain valid input, the ok button is disabled, and the user must fix the error. Also, there is error checking to confirm that the user does not enter a minimum value that is larger than the maximum value.

After, the input from the user has been validated, the process of creating the custom dictionary can begin. The first step in this process is to get the current dictionary, and the key-value pairs, where the key is the term, and the value is the frequency of that term, contained within the dictionary. The pairs are stored into an ArrayList, and an iterator is created to go through the entries of the ArrayList. While iterating, the value from the entry's key-value pair and the two frequencies the user entered are compared, and if the value passes the comparison, the term, and value are then added to a new dictionary. When the process is complete, the original dictionary is replaced by the newly created custom dictionary and the user interface updates to show the change. The difference between the two dictionaries can be seen in Figure 4.


Terms
◄ Prev 30 Next 30 ►
cognitive (923)
experimental (527)
sciences (512)
trends (501)
01 (500)
brain (328)
memory (248)
recent (247)
neural (222)
processing (199)
human (196)
visual (184)
learning (175)
studies (175)
perception (174)
processes (173)
research (172)
cognition (171)
information (169)
mechanisms (168)
social (166)
evidence (161)
language (155)
attention (138)
cortex (136)
understanding (129)
control (128)
models (121)
activity (108)
review (103)
Authors
Category
Cited
Date
Dbnumber
Doi

Before

Terms
◄ Prev 30 Next 30 ►
cognitive (923)
experimental (527)
sciences (512)
trends (501)
01 (500)
brain (328)
memory (248)
recent (247)
neural (222)
processing (199)
human (196)
visual (184)
learning (175)
studies (175)
perception (174)
processes (173)
research (172)
cognition (171)
information (169)
mechanisms (168)
social (166)
evidence (161)
language (155)
Authors
Category
Cited
Date
Dbnumber
Doi
Journal
Title
Year

After

Figure 4: The image on the left is the dictionary created by the parser and before the creation of a custom dictionary. The image on the right is after the creation of a custom dictionary, where the minimum frequency is 155.

4. Increasing Data Capabilities

The processing of large amounts of user data was the original intent of FutureLens. However, with some investigating FutureLens could not handle as much data as originally thought. As stated in the introduction, FutureLens has an upper bound on the amount of documents that it can handle. If the documents have an average size of four kilobytes then FutureLens can handle seventy-five thousand, and when the average size of the documents is forty-five kilobytes then FutureLens can only handle five thousand documents at one time. These numbers are low compared to the size of the data sets. For the documents with an average size of forty-five kilobytes, the ability to only handle five thousand documents allows the user to only process a small part of the collection at a time. Thus, leading to one of the main goals of this PILOT, how to increase the amount of data that FutureLens can process at one time [3]. The first step to solving this problem was to run a Java profiler [4] on FutureLens to see what was using the most memory. The profiler helped to determine that the top two memory users in FutureLens were strings and hash tables. Figure 5 shows the amount of memory that each of these structures use.

	Total size of hash tables in bytes	Total size of strings in bytes
500 – 2.1 MB	1897728	5061104
1k – 4.2 MB	3576512	10065200
2k – 8.2 MB	8213184	22906024
3k – 12.5 MB	12611328	35620576
4k – 16.4 MB	15723904	43210736
5k – 20.9 MB	18934688	53425800
10k – 41 MB	38432160	107248408
25k – 104.4 MB	89101952	257423216
50k – 209.3 MB	164210112	497489328
75k – 313.9 MB	232510272	715471040

Figure 5: The total size of hash tables and strings in bytes for the original version of FutureLens. This data was created using the psych abstract data set. The first column shows the number of files and the total size for that number of files.

These sizes seemed too large compared to the number and total size of the files being processed. The first step to increase the number of files that FutureLens could process was to investigate the hash tables being used in the program. There are eighteen hash tables in FutureLens, and all of them are keyed on strings. This lead to the discovery that if the number of hash tables can be reduced, the amount of strings would decrease.

The first place I looked to remove hash tables from FutureLens was during the parsing of the data set. Originally, FutureLens would read a document from the data set and when it finished would put the contents of the document into a hash table. When all documents were read into the hash table, FutureLens would then call the parser on the data set. This hash table of document contents would also be used by the graphical user interface (GUI) to display the document contents and any highlighted words that it might contain. After, discovering how the hash table was being used, I decided that this hash table could be removed. To remove this hash table, a document is now loaded and then parsed before FutureLens moves to the next document. This new procedure removes the need to store the contents of documents in a hash table. Since the hash table is removed, the documents are now read from disk when the user selects one to display. While removing this hash table I also removed hash tables from FutureLens's other data loaders, which removed an additional three hash tables. A hash table that was being created and populated during parsing was removed because other data structures where creating their own version of this hash table.

The total number of hash tables in FutureLens is now thirteen, which is an improvement when compared to eighteen in the original version of FutureLens. This reduction in the number of hash tables has also helped to reduce the number of strings being used. Decreases in the number of hash tables and strings has given FutureLens the ability to process larger data sets. It now seems, the only limit to the amount of data that can be processed is the amount of memory in the user's machine. On my machine FutureLens can now process two hundred thousand documents that have an average size of four kilobytes per document and handle ten thousand documents with an average size of forty-five kilobytes. Figure 6 shows the total size of the hash tables and strings in the new version of FutureLens with the changes discussed above.

	Total size of hash tables in bytes	Total size of strings in bytes
500 – 2.1 MB	1345120	3339496
1k – 4.2 MB	2596896	4929800
2k – 8.2 MB	6023552	9130040
3k – 12.5 MB	9312608	13240560
4k – 16.4 MB	11674368	16168048
5k – 20.9 MB	14055360	18838344
10k – 41 MB	28547008	36796640
25k – 104.4 MB	66867616	84901720
50k – 209.3 MB	126339648	158654776
75k – 313.9 MB	180497152	225158688
100k – 410 MB	241508064	301931832
200K – 829 MB	475282624	592066328

Figure 6:The total size of hash tables and strings in bytes for the new version of FutureLens. This data was created using the psych abstract data set. The first column shows the number of files and the total size for that number of files.

5. Decreasing Data Processing Time

The amount of time to process data becomes a factor as the amount of data FutureLens can process increases. The original version of FutureLens can process small data sets in a short amount of time. However, the time to process data increases rapidly as the size of data grows. Figures 7 and 8 show the average processing time for the original version of FutureLens. These figures lead to the goal of decreasing the amount of time FutureLens uses to process data.

File count/Size	Average Time in Seconds
500 – 2.1 MB	1.1595837
1k – 4.2 MB	1.5336145
2k – 8.2 MB	2.6333681
3k – 12.5 MB	4.4070547
4k – 16.4 MB	5.4321064
5k – 20.9 MB	6.302783
10k – 41 MB	13.2505653
25k – 104.4 MB	41.8450884
50k – 209.3 MB	129.678864
75k – 313.9 MB	259.3683904

Figure 7: The average time for the original version of FutureLens to process various amounts of psych abstracts across ten runs.

File count/Size	Average Time in Seconds
500 – 19.3 MB	6.5952379
1k – 40.7 MB	14.6556083
2k – 87.8 MB	38.3563106
3k – 131 MB	66.3107084
4k – 178 MB	100.9725318
5k – 218.3 MB	161.5056246

Figure 8:The average time for the original version of FutureLens to process various amounts of patent data across ten runs.

5.1 How Data Processing Time was Decreased

Data processing time was decreased by adding threads to FutureLens. The changes made to increase the amount of data that FutureLens can process helped to make it better suited for the addition of threads. Since, FutureLens now reads a document and then processes that document, the potential for threads to have an impact on the time to process data seemed greater than it did before any changes were made. Now, with an idea on how to decrease data processing time, the next obstacle would be how to actually implement threading in FutureLens. The first obstacle faced while implementing threads was my unfamiliarity with threads in the Java programming language. I had learned about them but never used them in practice. So after some research I decided to use Java's ExecutorService because of their ability to manage individual tasks and their ability to automatically use multi-core processors [4]. This second ability I discovered while experimenting with other ways to implement threads in Java, because the first way I implemented threads did not use the cores of my processor and this was noticed by looking at system usage. The next obstacle was to integrate the ExecutorService into FutureLens. This obstacle was resolved by creating an ExecutorService in the file loader and then creating a new class containing the document loading and parsing tasks as one function. The next problem was to find every variable that needed to be locked in order to stop race conditions between threads and produce results equal to those created by the original version of FutureLens. The solution was to follow a document through one iteration to find what variables change, and either change their type to a thread safe type, or add a lock around the parts of functions that modify variables. With these obstacles resolved the next task was to determine how many threads gave the best performance increase. Figure 9 shows the results of testing using various numbers of threads and the patent document data set.

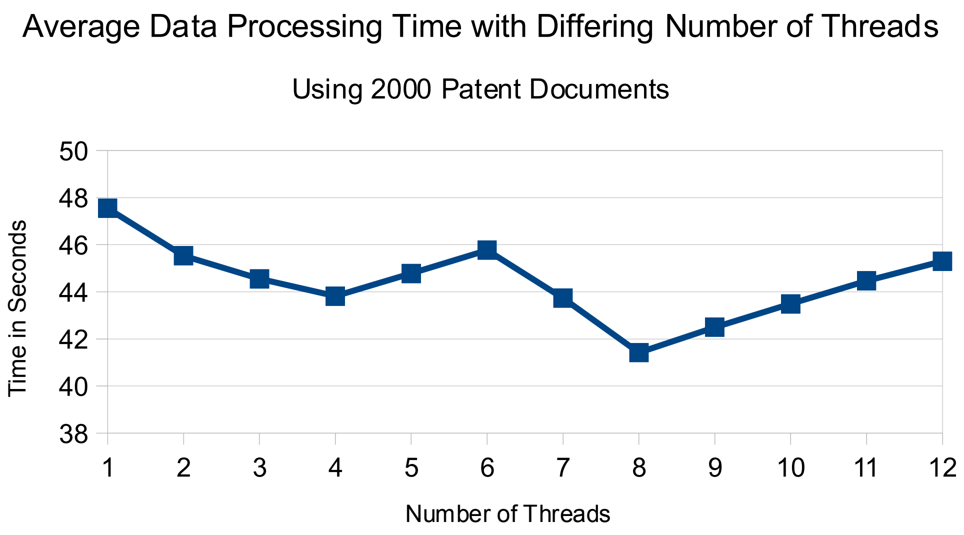


Figure 9: These averages are across ten runs.

As can be seen in Figure 9 the optimal number of threads is eight. This number happens to be two times the number of processors that Java says are in my machine. While testing, the upper bound increased to twelve, because I thought three times the number of processors would create a larger performance increase, after seeing the faster times at four and eight threads. However, this idea was proven incorrect and two times the number of processors is the amount of threads I decided to use.

The data sets used to test the original version of FutureLens were used to test the upgraded version. Also, larger data sets were used for testing because the upgraded version of FutureLens could handle them. The following figures show the average time for the upgraded version of FutureLens, and a visual representation of how the new version of FutureLens compares to the original version when using the psych abstract data set.

File count/Size	Average Time in Seconds
500 – 2.1 MB	1.3215038
1k – 4.2 MB	1.5722328
2k – 8.2 MB	2.6714057
3k – 12.5 MB	3.8262265
4k – 16.4 MB	4.6330552
5k – 20.9 MB	5.2030857
10k – 41 MB	8.6284829
25k – 104.4 MB	19.5933614
50k – 209.3 MB	36.9599291
75k – 313.9 MB	59.8540463
100k – 410 MB	69.7749802
200K – 829 MB	188.1068154

Figure 10:The average time for the upgraded version of FutureLens to process various amounts of psych abstracts across ten runs.

File count/Size	Average Time in Seconds
500 – 19.3 MB	8.499209
1k – 40.7 MB	18.5364414
2k – 87.8 MB	40.346865
3k – 131 MB	61.4941004
4k – 178 MB	83.7137996
5k – 218.3 MB	124.2625535
10k – 433.4 MB	236.2620091

Figure 11:The average time for the upgraded version of FutureLens to process various amounts of patent data across ten runs.

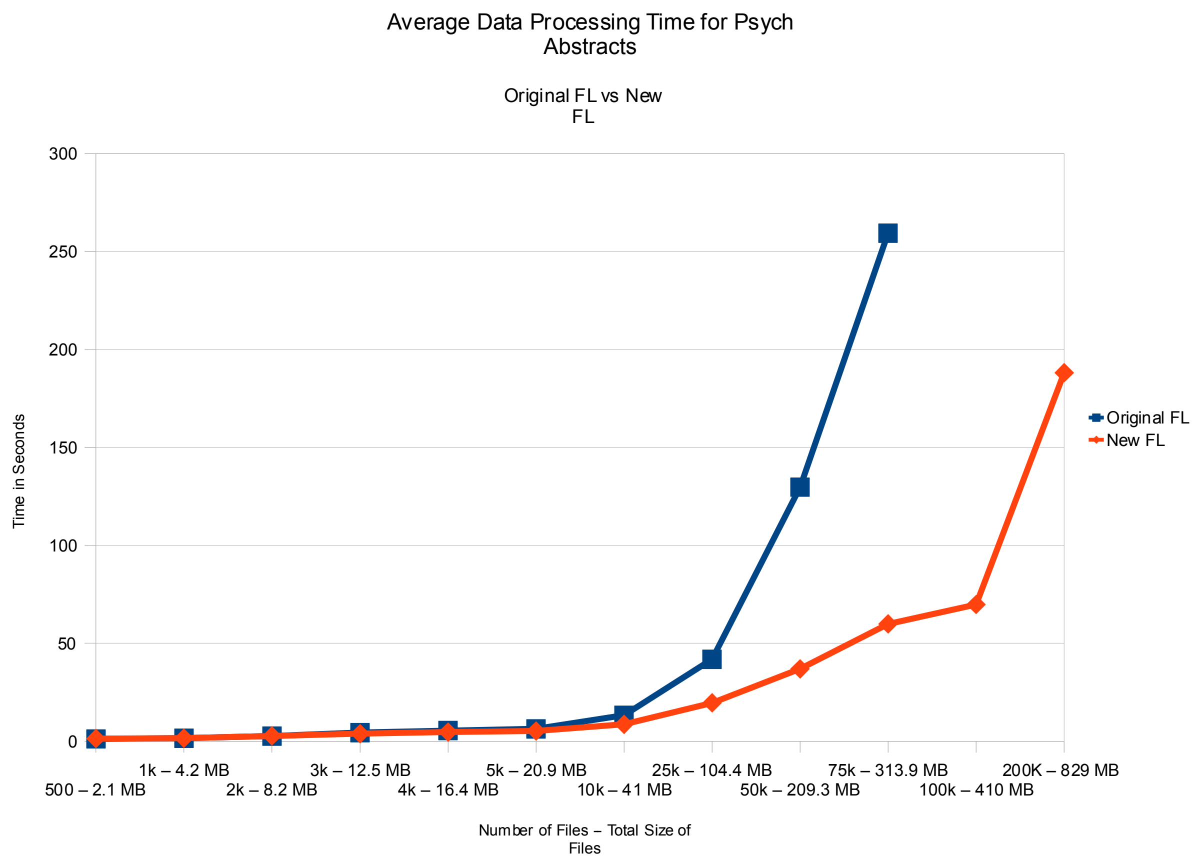


Figure 12: The average data processing time for the original version of FutureLens graphed against the average data processing time of the new version of FutureLens.

6. Future Work to Improve FutureLens

The first idea to improve FutureLens is to use a database. The database would hold the same contents that the hash tables currently hold, such as the dictionary, the inverted index, and the document names in the data set. The use of a database would allow FutureLens to run faster because the contents of these hash tables would not have be redone each time the application is used. To add the use of a database to FutureLens would not be that difficult. To store the dictionary, the application would make a database call instead of adding the value to a hash table, and for the inverted index it would be the same process. To make the process user friendly, the application would add all necessary information to the database so the user would not have to know how to use a database. The first time a data set is used, FutureLens would behave how it currently does by loading and parsing each document, with the only difference being the contents of the hash tables would now be stored in a database. If the user wanted to add more data to the data set, they select load files and the application would once again load and parse the documents, however it would only do those documents not currently in the database. The addition of a database would also allow for the user to continue working from a previous run of the application without having to wait for the data set to load again, which could be a huge time savings depending on the size of the data set. This addition could allow for FutureLens to handle data sets of any size both in average document size and total size in general.

The final idea for improving FutureLens is to write the application in a language where memory could be handled by the programer instead of the system handling the memory. The entire application does not have to be in a different language just the aspects that are memory intensive such as the hash tables, and the data structures. A different programming language could offer an improvement because the programmer knows when an object is no longer being used and should be deleted, however in Java the deletion of objects is handled by the system. I believe that if either of these ideas were implemented in FutureLens then there would be a large improvement in the overall performance.

References

[1] Gregory Shutt, Andrey Puretskiy, Michael W. Berry, “FutureLens”, Department of Electrical Engineering and Computer Science, The University Of Tennessee, November 20, 2008.

[2] "Eclipse documentation: ArmListner", http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fswt%2Fevents%2FArmListener.html, Visited February 2012.

[3] VisualVm 1.3.4, http://visualvm.java.net/

[4] "Executor Interfaces", http://docs.oracle.com/javase/tutorial/essential/concurrency/exinter.html, Visited April 2012