Artificial Teaching Assistant - Scarlet

ILHAN KARIĆ, ZANIN VEJZOVIĆ, DENIS MUŠIĆ, EMINA JUNUZ, MIRZA SMAJIĆ
Faculty of Information Technologies
University of "Džemal Bijedić"
Mostar,
BOSNIA AND HERZEGOVINA
ilhan.karic@edu.fit.ba, zanin@edu.fit.ba, denis@edu.fit.ba, emina@edu.fit.ba, mirza.smajic@edu.fit.ba

Abstract: - Scarlet an Artificial Teaching Assistant is a personal digital assistant that has been developed with main aim to assist students in their learning process by ensuring fast and efficiently search of documents and learning materials. Scarlet is able to give an adequate response to a specific question based on knowledge gathered by an unique algorithm which enables her to recognize context during file and web page content search. After finding the most appropriate answer Scarlet seeks for student feedback in order to improve future search. The metric proposed is based on the power law which occurs in natural language, that is the Zipfian distribution[1]. It is designed to work for any spoken language although it might work on some better than other depending on the nature of the language, the structure, grammar and semantics. The method uses this metric to derive context from data and then queries the data source looking for the best match. The whole implementation is rounded off by a learning module which gives the system a learning curve based on users (students) scoring how relevant the output is among other parameters. All the main algorithms and newly proposed metrics like the "contextual similarity" are presented in the same paper.

Key-Words: - Artificial intelligence, machine learning, pattern recognition, natural language processing

1 Introduction

Evaluating the similarity of two sequences is a standard computer science problem and has a wide area of use. A great deal of applications, from search engines to document ranking, from gene finding to prediction of protein functions, from network surveillance tools to anti-virus programs critically depend on analysis of sequential data. [1]

We will describe the matching of two sequences as well as the search process for the closest matching sequence from a pool of sequences which appeared to be the bottleneck of other methods like the Jaccard Index. We are also going to explain the naïve implementation which would directly follow from the mathematical model and optimizations, for which we have provided proof in the same paper, which will reduce the time complexity for both sequence comparison and sequence matching down to quasilinear time. In the end we will provide our benchmarks with data collected during the research.

2 The Problem

The main performance issue with sequence matching proved to be the fact that known methods didn't perform well on measurements made on the union of two or more sequences which are not directly correlated. This is mainly because those methods were designed to evaluate the similarity of sets, not sequences, meaning that the data must be either sorted in some way or split up into smaller logical chunks to reduce the time required to find the closest match. Even if the initial similarity function was linear, the search for the closest match would have to be quadratic because the function would have to be performed for each set individually. The contextual similarity function proved to be of quasilinear-time complexity for both comparison and matching of sequences.

3 The Function

The basic idea behind the concept lies in one of the most important properties of a sequence, the order of items within it. The second property, the content, will be awarded in a way where it won't matter as much as the order or position

where it is located. The best idea is to imagine the initial sequence as a set of characters (sentence) and the second as a text. The goal is to score how relevant the text is given the first sequence of characters. From this point on, the first sequence (sentence) will be denoted as N and the second (the text) as M.

The probability of *N* being found as a subsequence of *M* due to sheer coincidence (without having any contextual relation with it) decreases exponentially as the length of *N* increases. We will apply this as a heuristic although it has some deep roots in linguistics due to the power law and the Zipfian distribution. [2] This means that we're going to award the appearance of a subsequence of *N* within *M* based on the length of the subsequence naming it shared context:

$$\delta(N,M) = \sum_{i=0}^{|N|-1} \sum_{j=i+1}^{|N|} |N[i:j) \cap M| \quad (1)$$

Note that $N[i:j)$ represents the substring of *N* starting at position i (inclusive) and ending at position j (exclusive), $|N|$ represents the total length of the string. We used the intersection symbol $\cap$ to denote if or if not M contains the entire substring $N[i:j)$. In case it does the length $|N[i:j)|$ is added to the score, 0 otherwise.

We are summing up the lengths (cardinalities) of all possible subsequences of *N* found in *M* thus rewarding the position and order rather than the actual number of matches. We don't pay attention to how many times a certain subset occurs to avoid "is", "the", "a" and similar words to add up on quantity rather than the way they create contextual meaning.

The problem with this measurement is that it has no upper bound thus grows infinitely. This would result in sequences which are generally larger to be "more similar" than other sequences with a smaller length which is not true. To solve this problem we must normalize our function by dividing with the coefficient T_δ which represents the total score that can be achieved under the assumption that the first sequence is a proper subsequence of the second meaning that $N \subseteq M$. In this case we can compute T_δ directly as shown in (2) below:

$$T_\delta = \sum_{k=1}^{n} \frac{k(k+1)}{2} = \frac{n(n+1)(n+2)}{3!} \quad (2)$$

Assuming that *n* is replaced by the length of our first sequence *N* we can express our normalization coefficient as the following binomial:

$$T_\delta = \binom{|N|+2}{3} \quad (3)$$

Now, once we have normalized our initial function (1) we arrive at the final expression for the contextual similarity:

$$\bar{\delta}(N,M) = \frac{1}{T_\delta} \sum_{i=0}^{|N|-1} \sum_{j=i+1}^{|N|} |N[i:j) \cap M| \quad (4)$$

This is the normalized function and measures the contextual relation between two sequences.

The domain of the function (4) is $0 \leq \bar{\delta} \leq 1$ where the similarity (and probability of non-random relationship between the two sequences) is increasing with $\bar{\delta}$ respectively. When the entire first sequence is a subsequence of *M*, the contextual similarity $\bar{\delta} = 1$. In the case where the two sequences are disjunctive, $\bar{\delta} = 0$

4 Optimizations

We will stop to review the naïve implementation which would follow directly from the mathematical model. The first thing to notice is that if our second sequence M does not contain a subsequence $N[i:j)$ there is no need to check $N[i:j+1)$, if N = "ABC" and M = "AXBC" we will find that M does not contain "AB" thus it is redundant to check for "ABC" and we can stop further computation on the given subsequence of N.

The second optimization addresses the problem where $N = M$ or in general when N contains long subsequences of M resulting in many nested iterations. If we look closely we will notice that our normalization coefficient can be used in a way which will lead to us avoiding re-doing done work. Consider the following example where N = "AAAAxA" and M = "AAAAAA". Once we've evaluated the sum of all lengths of all subsequences of N up to "x" ("AAAA") we can agree that repeating the same process for ("AAA") is a waste because we know that they exist within M so we can skip that part and add $\binom{|N|-1+2}{3}$ directly. This will further reduce the gaps between the worst, average and best cases.

The third optimization would be to use the unique property of the contextual similarity function that allows us to review two or more sequences at once by concatenating them. These sequences can be picked at random and don't have to be correlated, meaning that we could merge four sequences into two super-sequences and review two sequences at once. Note that we should add one character as a separator to avoid creating subsequences which weren't there initially. For this we should use a character which is not part of the alphabet. Now, we can compare our initial sequence N with, for example, two super-sequences $M_{p1} + M_{q1}$ and $M_{p2} + M_{q2}$ in two computations rather than four. The super-sequence which scores more has an increased probability of one of its subsequences to be related to N. Once we decide which super-sequence probably contains our information, we can slice it in two halves and repeat the process until only one item is left. To keep the information about the initial groups we will use a list where each item is a known sequence M and only do grouping logically, based on indexes, performing a context-driven dichotomic search. This way, instead of doing 1024 computations for 1024 sequences we only have to do $2\log_2(1024) = 20$.

5 Theoretical Complexity

In this section we're going to discuss the worst, best and average time complexity of the optimized algorithm.

The worst case occurs when the two strings we are comparing are equal. In this case the outer loop would only iterate once (due to the second optimization) and the inner loop would iterate N times. For this calculation, we're assuming the Boyer-Moore string search algorithm which is known to have a linear average time complexity. This would give us the worst case time complexity of $O(1 * n * n) = O(n^2)$ so we can say that our algorithm, in the worst case, will execute in quadratic time.

The best case occurs when the two strings that we are comparing are disjunctive thus the outer loop would iterate N times while the inner loop would iterate once per outer iteration and the string search operation would be performed on a single character every time giving us the best case time complexity of: $O(n * 1 * 1) = O(n)$

The average case, unlike the previous two, is more difficult to evaluate. Since we don't know how the input will look like, we're assuming two random sequences each of length N. The actual character at any position is picked uniformly at random from an alphabet of size q. The probability of picking any specific character is $p = \frac{1}{q}$. By picking two fixed positions, we can say that the probability for m consecutive matches, starting at position i in the first sequence and starting at position j in the second sequence is p^m.

Let $x_{ij}(m)$ be a random variable with the value 1 if starting at position i and position j there is a match of length m and 0 otherwise. It takes on 1 with the probability p^m and 0 with the probability $1 - p^m$. Counting all matches we can say that:

$$C_m \equiv \sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij}(m) \tag{5}$$

Remember that C_m represents the total number of pairs with starting positions i, j that have a match of length m. We're interested in the expected average value of $E(C_m)$ because that will tell us what the most likely length of the common substring of two randomly generated strings is. We can imagine this as taking the mean over infinitly many lengths of the longest common substrings for two randomly generated sequences of length m. This is expressed in the equation (6) below:

$$E\left(\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij}(m)\right) \tag{6}$$

Due to the linearity of expectation we can take the sum of the expected values instead of the expectation of the sum so we can define $E(C_m)$ as:

$$E(C_m) = \sum_{i=1}^{n}\sum_{j=1}^{n} E\left(x_{ij}(m)\right) \tag{7}$$

If we look at the expectation of $x_{ij}(m)$ we can see that

$E\left(x_{ij}(m)\right) = 1 * p^m + 0 * (1 - p^m) = p^m$.

We can finally express the expected number of matches in two random sequences as shown in (8):

$$E(C_m) = \sum_{i=1}^{n}\sum_{j=1}^{n} p^m = n^2 p^m \tag{8}$$

The assumption is that $n \gg m$ which means that we don't have to account for different upper limits of the two sums. This is how the algorithm was designed to work so it is a good approximation for the average case. We're interested in how $E(C_m)$ behaves asymptotically as m increases.

Let r be the largest value of m such that $E(C_m) \geq 1$. Knowing what r is will give us a good heuristic insight into knowing what the expected longest common substring is. We arrive at the following inequality:

$$n^2 p^r \geq 1 \tag{9}$$

Since r is the largest value of m such that this holds, we must also say that $n^2 p^{(r+1)} < 1$. Once rearranged and further simplified, we express the probability p in function of the

alphabet size q as initially and arrive at the following approximation:

$$2\log_q(n) - 1 < r \leq 2\log_q(n) \qquad (10)$$

From this approximation we can learn about the asymptotic behaviour of the expected length of the longest common substring as the length of the string n increases. A simulation has been created to help visualize our mathematical predictions which can be seen in Fig. 1. We have created pairs of randomly generated sequences of the same alphabet size and measured the longest common substring length in function of the lengths of the two sequences. Each data point on the scatterplot in Fig. 1. was computed as the arithmetic mean over 1000 iterations in order to rule out random occurances as much as possible. We can see the results of this simulation the scatterplot below:

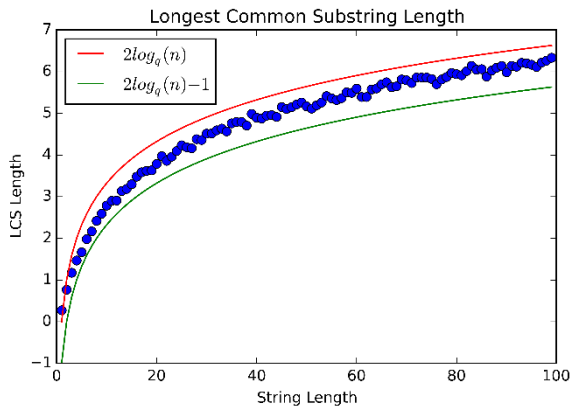

Fig. 1. The asymptotic behavior of the growth of the longest common substrings in function of string size.

We can see our upper and lower bounds which give us an insight into where we expect to find our random values.

With this approximation proven, we can finally compute the average time complexity of our algorithm. To simplify and leave some room for error, we're going to take out the second optimization from our evaluation. This means that the outer loop structure will always iterate N times while the inner loop, due to the first optimization, only iterates in function of the longest common substring which was just proven to grow logarithmically. Since the string matching algorithm inside the second loop only ever operates on the longest common substrings

of length $\log(N)$ we can say that the average time complexity of our algorithm is:

$$O(n * 2\log n * \log n) = O(n\log^2 n)$$

It is important to note that our method allows for measurments on the union of two or more sequences. Since we know that binary search has a known average time complexity of

$\log(m)$ (where *m* is the total number of items) and saying, for the sake of simplification, that

$n = m$, meaning that the number of sequences to search through is equivalent to the number of symbols in each sequence (which is way more than the usual scenario of use) the complexity of comparing our sequence to all existing

sequences is: $O(n\log^2 n \log n) = O(n\log^3 n)$ that is, still quasilinear. In contrast, other methods such as the Jaccard Index would perform search operations in quadratic time thus the proposed method is faster.

Another thing to note is that the reason why our assumption of random strings would be a good representative for the average case is due to the fact that both randomly generated and coherent text follow a Zipfian distribution. [2]

6 Benchmarks

We have measured only the first and second optimization impact on the general performance of the algorithm. The third optimization was always enabled to speed things up. The first three tests have been ran for all three cases (with no optimization, first optimization and second optimization enabled). The first sequence N was always populated with 33 characters and each list item M was exactly 33 characters long. The list was filed with 2, 4, 8, 16.. 32768 items where every item was 33 characters long. The first test populated the list

with items which would yield a very high similarity when evaluated. The second test generated a list of random items each 33 characters long, to test the performance against random similarity. The third test populated the list with items that would yield a low similarity on average.

Our theory predicted that the first optimization would reduce the execution time when the similarity was high (worst case) and that our second optimization would further reduce the speed gap between the worst and best case (high and low similarity). The results only show how each optimization changed the algorithm performance in the case of high, random and low similarity in data.

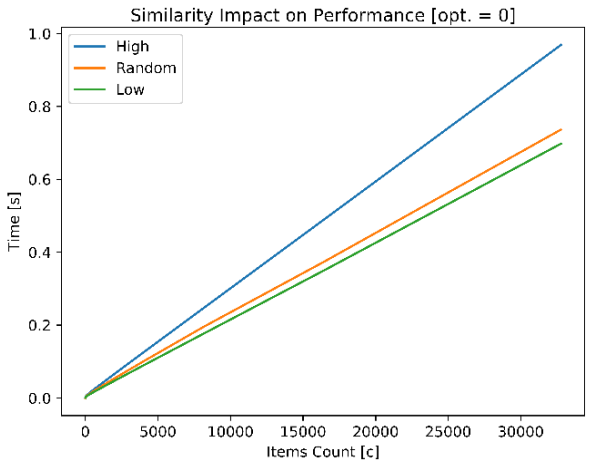

Fig. 2. Algorithm search performance in the case of low, random and high similarity data, without optimizations.

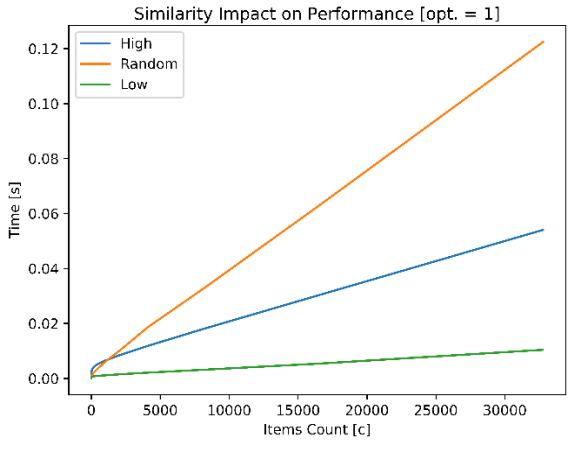

Fig. 3. Algorithm search performance in the case of low, random and high similarity data, with only the first optimization enabled.

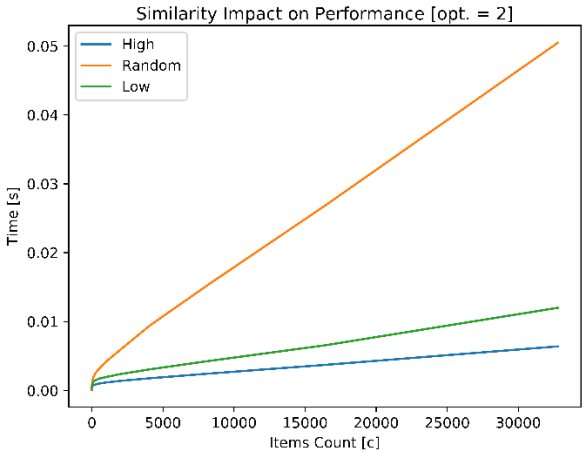

Fig. 4. Algorithm search performance in the case of low, random and high similarity data, with only the second optimization enabled.

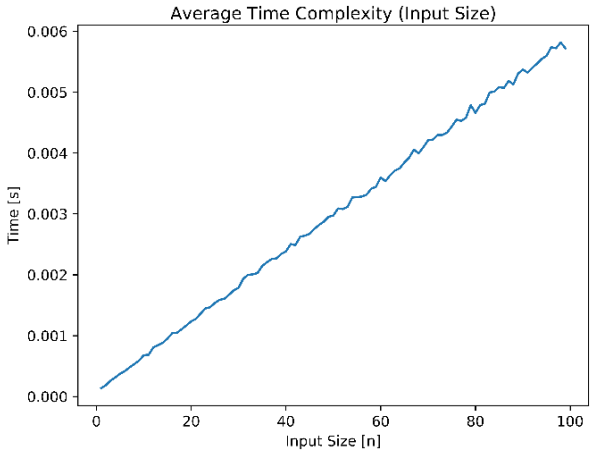

Fig. 5. Asymptotic behaviour of the average time execution in function of the length of the first sequence.

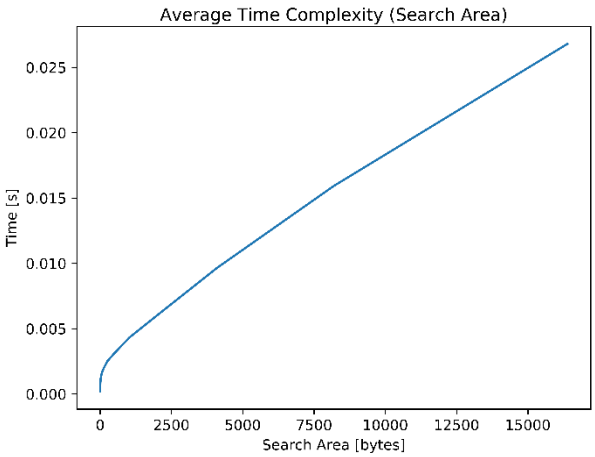

Fig. 6. Asymptotic behaviour of the average time execution in function of the number of sequences to

search through. All sequences have been generated randomly.

7 The Scarlet System

Scarlet is designed to work best with a question / answer scheme but is highly adaptable to a vast range of different input schemes. Complete system can be divided into 3 mayor modules: *natural language processing*, *pseudo contextual data analysis* and *trial & error learning*. Complete Scarlet system and relationship between these modules is presented in the Figure 1.

The *natural language processing module* implements different methods trying to model and approximate answer patterns based on the input pattern. This is a very important first step because probability of the search method to find a solution/answer is increased exponentially depending on the quality of the parsed input. This module is in a feedback relation with the trial & error learning module.

The main component of the Scarlet system is the *pseudo contextual data analysis* which makes the same implementation capable of working for almost any spoken language. This module takes the processed input data from the previous module and recursively browses through all the folders and sub folders on a FTP or local server. The currently supported document files are "doc", "docx" and "pdf".

To speed things up the end user (student) can beforehand apply a filter mask by selecting a class subject. In addition, this module supports web search. It allows the user to search the web (up to a certain depth) and process the information from web pages, prompting what is most likely the answer for the given input.

The *trial & error learning module* is the last of the three modules. At this point the user is prompted a question if the Scarlets' answer was good or bad. Depending on this user feedback, in case it was positive, the system saves the given input-output pattern into the pattern table. The next time when the user sends some request to the system it will first browse through this table and compare the user input to the known inputs. When the system finds result that is "fit" enough it will look at the saved input-output pattern table and try to find how to morph the input in order to get the desired output and applies this strategy to the new user input. The algorithms used in aforementioned process will be described later in the paper.

8 The Scarlet Algorithms

As mentioned in previous part of this paper, there are a few algorithms that were developed for these specific purposes. The two most notable ones are: *the pseudo contextual data analysis algorithm* which will be described in detail.

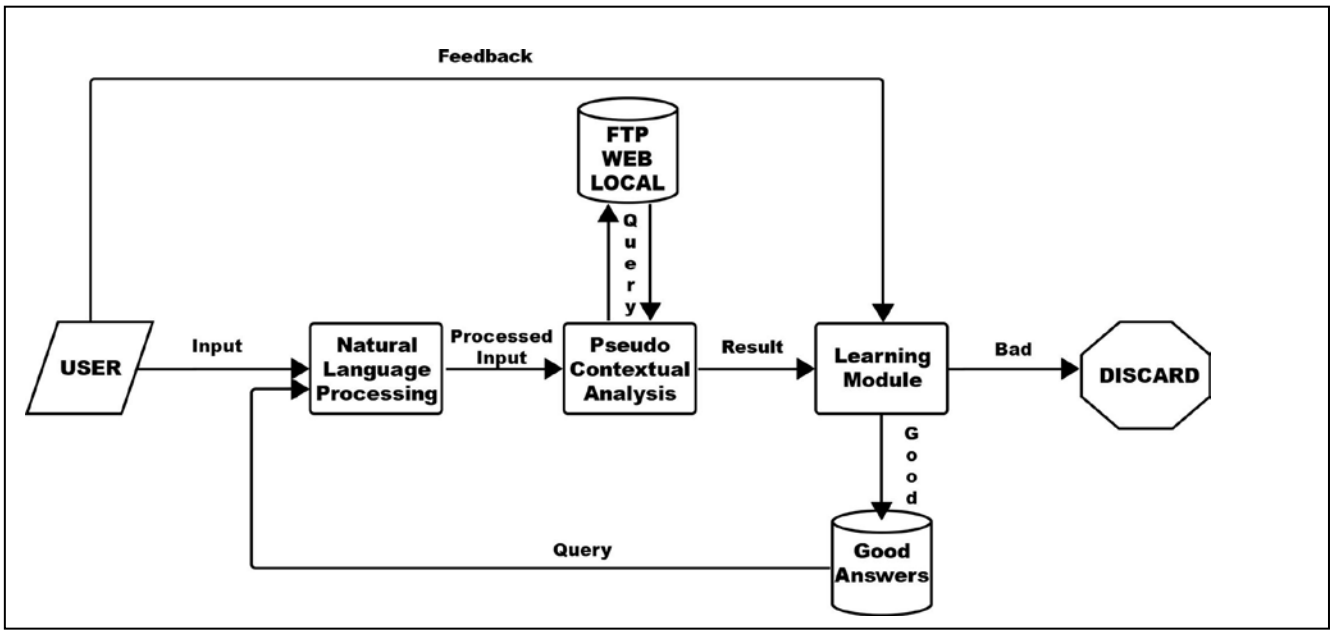

Fig. 7. Scarlet System Components

8.1 Pseudo contextual data analysis

This algorithm is the main algorithm within the pseudo contextual data analysis module (contextual similarity[1]). The goal of this algorithm is to determinate similarity between two sets (S1 and S2). Similarity determination is done by the previously presented contextual similarity equation (4).

The algorithm is divided into two steps. The first step is to run and compare the entire document file to the given input string. We do this for all documents and only take the one with the biggest score (most likely containing what we need) and prompt the student which document probably contains his answer and should be appropriate for studying.

During second step algorithm parses all sentences from the text as individual strings and repeat the algorithm for each sentence. We keep only the one sentences that scores the most thus most likely contain the definition or answer that user is looking for.

An additional (optional) third step can be applied in terms of a selection filter which allows user to narrow down the searching area by defining subject to his question belongs to.

8.2 Structure Anchoring Algorithm

Structure Anchoring Algorithm is the main algorithm in the *natural language processing module*. It is also used in the *trial & error learning module*. For three given strings (S1, S2, S3) it will try to "morph" S1 based on how you can construct S3 from S2. It does this by finding the biggest common substring found in S1, S2, and S3. A suffix tree can be used to find substrings fast and efficiently. This depends on how large the string is (i.e. if it pays off to construct a tree first) and then the transversal is in O(n) linear time.

During The next step system anchors this common information and observes everything left and right of the anchor point. First it anchors the sample input S2 and the sample output S3. Then it looks for words (left of the anchor) that have been moved right of the anchor. If it finds only one word that does this, it anchors string S1 and moves everything left from S1's anchor point to the right. The same algorithm is applied for words that have been right of the anchor point and which migrated to the left.

Here is an example situation where Structure Anchoring is applied.

[1]In this paper word *sets* will be considered as synonym for partially ordered sets

S1 = "Who is Andrew?"

S2 = "Who is Robert?"

S3 = "Robert is a person"

The anchor in this example would be the word is because that's the only word common in all 3 strings at once. Next it looks if any word left of the anchor point in S2 is contained right of the anchor point in S3. This is not the case so it moves on checking for the opposite case (if any word right of the anchor point in S2 is contained in S3).

The word "Robert" fulfills these criteria thus everything that is right of the anchor in S1 is moved to the left. Finally, system joins these strings as follows {newLeft} {anchor} {new Right} to get the newly morphed S1: "Andrew is".

It is important to note that there is no way to turn the string into "Andrew is a person" because system at this point is unable to conclude that the word Andrew has any contextual similarity with the word Robert. However, its content is good enough for the pseudo contextual search algorithm which will most probably produce better result.

Although the algorithm seems rather simple and inefficient, it is able to increases the rate at which the system finds an appropriate response. A neural network would probably be a better solution and produce much better learning curve.

9 Implementation

In order to demonstrate the practical application of the developed algorithm we implemented basic version of Scarlet agent shown in Figure 2. Current version contains three main operating modes: Web, FTP and Utility.

In the first operating mode (Web) agent is configured to search the Internet. After the user enters a specific question agent permutes the input string so that it can recognize the form of the appropriate answer, for example:

>> Input: What is the capital of France?

>> Output: The capital of France is

This is followed by a second step where newly transformed string is entered into Google's search engine which returns the first N websites. Result set is then parsed by the pseudo contextual analysis algorithm which calculates how similar the transformed string is to the text that is received from Google. When the best source of information is determined, the same algorithm is applied to all sentences within the best results and return a sentence that have the highest score.

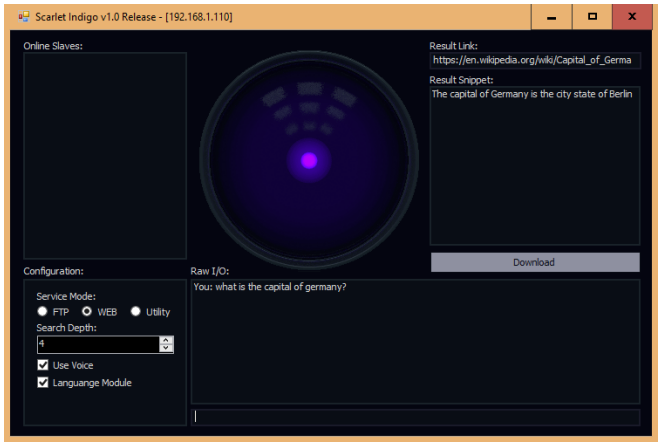

Fig 8. Scarlet System Components

FTP operating mode uses the same principle but with a difference that files represent source of information and are placed on the FTP server. It is important to emphasis that the used algorithm works the same for any language.

Utility operating mode is basically the same algorithm that compares the input string with fixed template string and based on "context" execute various commands like opening or closing programs, searching YouTube etc.

10 Conclusion

This Scarlet system proved to be very effective in contextual information analysis thus in returning the proper document that is adequate to the subject. It can differentiate between different subjects (Databases, Programming, Communication Technologies etc.) due to the

different context and keywords. The case where the system has found not only the right subject, but the right lecture is also very high. It mostly returns the exact definition the user asked for (not using the learning module which would yield even better results).

References:

[1] Konrad Rieck, Pavel Laskov, “Linear-Time Computation of Similarity Measures for Sequential Data”, Journal of Machine Learning Research 9 (2008) 23-48 pp. 1

[2] S T. Piantadosi, “Zipf’s word frequency law in natural language: A critical review and future directions”, Psychonomic Bulletin & Review, vol. 21, 2014,pp.1112-1130 to perform linearithmic due to the nature of the function.

[3] Ljubomir Lazic, Nikos Mastorakis, "Cost effective software test metrics", WSEAS Transactions on Computers, pp. 599-619, Vol.7, Issue 6, 2008

[4] Klimis Ntalianis, Nikos Mastorakis, "Social Media Video Content Diversity Visualization", International Journal of Signal Processing, pp. 169-176, Volume 1, 2016,