# Rivet: Automated Data Analysis Using The Rosie Pattern Language
# Final Project Report

Requirements, Design, Implementation/Testing & Installation/Delivery

Jamie Jennings, Ph.D./IBM

CSC 492 Team 11:

James Baggs
Xiaoyu Chen
Yuxu Yang

North Carolina State University
Department of Computer Science

December 13th, 2017

## Executive Summary

Author(s): James Baggs

IBM estimates that more than 2.5 billion GB of data is produced daily, and 99.5% of it is never analyzed. This data can hold valuable information about the people, systems, or processes which produced it and extracting this information programmatically is a source of ongoing work and research. As such, our team has been tasked by IBM to work with Dr. Jamie Jennings and the Rosie Pattern Language (RPL) to create an efficient solution to extract information from semistructured data.

The Rosie engine is a promising improvement over the industry standards for data extraction and pattern matching, such as grok and regular expressions, but its standard operation relies on brute force matching to categorize data into known data types and extract information. Knowing which patterns of data types are most likely to be present in a file can greatly reduce the time required to match data to its type. When provided with knowledge of datatype patterns within a file in the form of an RPL expression, Rosie is able to parse data up to three times more quickly compared to its brute force approach.

Our team's task is to remove the need for a human to manually write these RPL expressions, and instead produce them programmatically in such a way that Rosie can use them to parse data at a higher speed while still matching the same range of datatypes that would be matched by its brute force.

The challenges we have faced with this task include the familiarization with new technologies, the creation of efficient data extraction patterns, and error handling of non-uniform or missing data. Our team started out unfamiliar with both Python and Rosie and had to learn the languages and their functional usage in order to implement an efficient protocol for extracting patterns from a file generating and RPL. The system was revised several times to improve its efficiency and ensure the correctness of its automatically generated RPL patterns.

Rivet is capable of interacting with the python library from version 0.99k of Rosie and producing a pattern analysis of the data in a given semi-structured data file. Rivet accepts command line arguments for input file sampling percentage, an output pattern prune percentage, and a file to produce the RPL pattern for. Rivet uses Rosie's brute force engine to match data types in the sample file, and then uses this information to extract line-patterns (patterns of datatypes within one line of data). In user-interactive mode (no prune percentage specified) the system prints these line-patterns to the console so that a user may specify which patterns to combine and save for data extraction. The also contains tests to verify that the parse speed of a file in Rosie given either user-generated or automatically generated file patterns meet our requirements for speed improvement over the brute-force approach.

We designed development iterations to include incremental feature-based improvements to the Rivet system, starting with a brute-force wrapper to the Rosie system, and ending with a user interface and end-to-end test cases. After these iterations, Rivet now provides users with tools to generate RPL patterns which extract data up to 30% faster on data which is non-uniform (~100 line-patterns per file) and over two times more quickly on relatively uniform semistructured data (~4 patterns per file).

# Project Description

## Sponsor Background

Author(s): Xiaoyu Chen
Minor Edits: James Baggs

IBM (International Business Machines Corporation) is a multinational technology company which manufactures and markets computers hardware, middleware, and software. It is also a major research organization. Our Sponsor for this project is Dr. Jamie A. Jennings, a Senior Technical Staff Member of IBM Cloud CTO Office and the Developer of Rosie Pattern Language. She focuses on finding solutions to data mining problems, increasing efficiency and reducing the amount of human work needed to analyze data.

Name: Jamie A. Jennings, Ph.D.
Email: jjennings@us.ibm.com
Phone: 919-491-6751
Address: 3039 East Cornwallis Road
Research Triangle Park, NC 27709

## Problem Statement

Author(s): Xiaoyu Chen
Editor(s): James Baggs, Yuxu Yang

Billions of gigabytes of data are produced daily, and most of it is never analyzed. It can be difficult to extract useful information from large data sets especially no matter they are unstructured or only semi-structured. Data from diverse sources may be formatted in unique structures and patterns in comparisons across sources, and also in comparison between lines from one file. While it might be expected that a CSV file exported from a program like Excel should be uniform, this is simply not the case for the majority of data. Regular expressions are currently used to identify data and analyse information from these data sources, but the complexity of Regular expression resulted in poor readability and maintainability. Regular expressions are inefficient both in terms of its user-friendliness (minimally composable patterns, complex format), and its computations (high backtracking, cpu cache unoptimized)

RPL was designed to be an alternative to regular expressions. It makes use of defined patterns to write its expressions. However, like regular expressions, Rosie must iterate through every pattern in its library and try to find a match for each token of data. To match data more efficiently, it is important to programmatically leverage knowledge about the structure and repetition of data types within a file, in order to reduce the amount of time for native brute force computation.

Since RPL is a specific language for identifying large data sets, the original way to use rosie to analyse data is still time consuming. Our project is designed to add layers based on Rosie engine, which can generate specific RPL pattern for different files and use that to analyse different large semistructured data sets repeatedly. Using targeted RPL pattern for certain semistructured data sets repeatedly, the amount of time of analysing data will be reduced at least 30% compared to rosie's native brute force computation.

## Project Goals & Benefits

Author(s): James Baggs
Editor(s): Yuxu Yang

The goal of this project is to develop a more user-friendly system which will analyze large quantities of raw, semi-structured data faster and as accurately than Rosie's brute force approach. This means that we aim to extract data twice as fast for uniform data (< 5 line-patterns per file) and 30% faster for non-uniform data ( >100 line-patterns per file) while providing the same JSON output that Rosie would produce by default. We additionally want to make the system more user friendly, particularly to those who are unfamiliar with regular expressions or the Rosie Pattern Language, by providing a clear, concise data extraction report and removing the need for a user to know how to code in RPL altogether.

The Rosie Pattern Language is accessed through the command-line interface, and it requires that users specify patterns to match within the chosen file. The intended implementation of Rivet would automatically analyze a file and generate a set of patterns for a user to pass to Rosie to parse data with, and then later provide a report to the user which describes the types and frequencies of line-patterns within a file. This is unique from Rosie's implementation, which only outputs parsed data in a complex and nested JSON format that is computer-friendly, but difficult for humans to read. This JSON format nests data into its parent datatypes as defined by rosie. Rivet is designed to extract the most specific child pattern that a token matches to because this is the most useful information for a user because it most accurately identifies what the token is. For example, a token identified as a "word" is too general, but it is more useful to know that the token is an URL.

A system which achieves these goals would greatly reduce costs associated with data mining. For large computations, such as the work Rivet/Rosie are designed for, the cost of operation becomes roughly proportional to the amount of computational time. A system which is analyzing and extracting billions of gigabytes of data would require large amounts of processing power, and could cost many millions of dollars. A system such as Rivet which cuts the extraction time in half would almost certainly reduce the data extraction cost by millions as well. Additionally, reducing the amount of human labor required to analyze data files and produce RPL patterns also reduces labor costs. While our system was mostly focused on addressing efficiency problems, the techniques used and the data reports provided to users could additionally benefit data analysts by acting as a training tool for Rosie. Being able to see how efficient line-patterns can be constructed given a set of sample data could benefit a user's understanding of how to construct their own efficient RPL expressions for data which is more specific and cannot be auto-matched by Rivet.

## Development Methodology

Author(s): Yuxu Yang
Edited by: James Baggs

The team is using an iterative development process where each iteration lasts 2 to 4 weeks. There will be about 4-6 iterations in total. Currently, the team meets with the sponsor every week. The team anticipates that requirements will change over the course of the semester which makes the iterative process an appropriate choice for this project. Each iteration typically follows the process below:

- Examine iteration requirements
- Create a preliminary iteration design
- Meet with sponsor to validate design or elicit requirements
- Split design and requirements into concrete tasks
- Assign different tasks to team members
- Implement tasks
- Meet with sponsor for implementation demonstration
  - If the sponsor validates the demo, elicit design for next iteration
  - Otherwise elicit more requirements and extend current iteration
- Create tests for validated implementation
- Update design documents

## Challenges & Resolutions

Author(s): Yuxu Yang

The first challenge we had was to figure out how should we target individual patterns in each line of data set to against the existing patterns in Rosie's library. After much deliberation with Dr. Jennings, we had defined the concept of file-patterns. File-patterns are a set of smaller patterns which generated by taking a sample of the given semi-structured data and matching the lines of data against Rosie's library of pattern.

A significant challenge is sampling. One of the goal's for Rivet is to be faster and more accurate. A larger sample size allows Rivet to be more accurate but slower. A smaller sample size allows Rivet to be faster but less accurate. Research, experimentation and Dr. Jennings' help are needed to find an acceptable threshold. Later on in this semester, we developed a sampler to generate random data to be analysed.

Another significant challenge is the processing time of our program. The Rivet system spend huge amount of time to analyse huge data set such as 100k lines of semistructured data during the first and second iteration. Later on this semester, we used Json object to pass the huge amount of data without spending on I/O reading and writing. By using this way, we reduce tremendous processing time.

For Rivet to work faster, the team decided that file-patterns should be ordered by most frequently-occurring to least frequently-occurring. Dr. Jennings introduced the idea of using a tree data structure. The team decided on using a linked list tree data structure because of the use of nodes. Each node holds a counter to count how many times each file-pattern occurs.

Another way to make Rivet work faster is to keep the list of file-patterns as small as possible. Dr. Jennings suggested that insignificant file-patterns are pruned from the tree. The team has defined an insignificant file-pattern as a file-pattern that meets a defined low frequency threshold. For example, an insignificant file-pattern may be any file-pattern that occurs 2% or less out of all the file-patterns. The decision to prune the tree also depends on how well the sample represents the data. For example, if the sample is bad, a file-pattern that was determined to occur 2% of the time may actually make up 25% of the full semi-structured data.

## Resources Needed

Author(s): Xiaoyu Chen, Yuxu Yang
Editor: James Baggs

| Resource Name | Version | Description | Obtainment Status |
|---|---|---|---|
| Python | 2.7.2 | Python programming languages support our Rivet system to meet the design requirements | Yes |
| Rosie | 0.99k | Rosie Engine is designed to analyse large sets of semistructured date. | Yes |
| Linux terminal | | Since Rosie engine only works on Linux terminal Rivet can only work on Linux terminal | Yes |

The most important resource needed for this project is Rosie Program Language. Rosie serves as a user-friendly alternative to Regular Expressions (regex), and is designed to extract information from data. Rosie is an open source software, and the the version that is currently used by the Rivet system is v0.99k (v0.99 branch on github). Rosie v1.0 is available now, but the libraries and engine needed for Rivet are not yet available.

Rivet is developed by using python 2.7 language, thus in order to run Rivet user must have python version 2.7.2 or higher installed. Instructions for installation of python 2.7 are provided in the Rivet Installation guide.

Since Rosie engine only works on the Linux Terminal, thus, in order to run rosie, user must have the access to a Linux terminal from a Linux machine, or mac terminal, or windows 10 Anniversary bash, there are also instructions on how to install windows 10 Anniversary bash in the Rivet installation guide.

## Requirements

Author(s): Xiaoyu Chen
Edited by: James Baggs

The Rivet system uses the Rosie Pattern Language's existing library of patterns to create new customized patterns for semi-structured data. Rivet gives users the ability to select a sample of a data set, then analyze all possible patterns and determine the frequency of each pattern's occurence within the sample data file. Users have the freedom to select different sample sizes of the original data file, user also is able to provide a prune percentage for Rivet to automatically generate a RPL file that contains all patterns that are above provided prune percentage. On the other hand if user do not provide a prune size, Rivet will ask user to select patterns and a new name for RPL file for the final generation.

A file-pattern is generated from a line of sample data, and it is composed of smaller patterns that were identified by matching the pattern in each part of a line of sample data to the patterns that already exist in Rosie's library. The brute force method refers to Rosie's ability to match patterns against its own library of defined patterns.

1. Functional Requirements
    - 1.1. Rivet shall be accessed via a command line interface.
        - 1.1.1. Rivet should have command line options for sample size and prune size for user(-s for sample size, -p for prune percentage)
        - 1.1.2. Rivet should able to take a filename as an argument
    - 1.2. Rivet shall accept semi-structured data that are already formatted into lines.
        - 1.2.1. The system should accept CSV format files as input.
        - 1.2.2. The system should accept HTML format files as input.
        - 1.2.3. The system should accept plain text format files as input.
    - 1.3. Rivet should produce a sample that accurately represents the data with correct sample size provided by user.
    - 1.4. Rivet should automatically generate pattern and RPL file if user provide a prune percentage when running Rivet
    - 1.5. Rivet shall generate a JSON file for output for sample data that includes patterns.
    - 1.6. Given that Rivet has generated RPL file from a set a sample data, the system shall parse the semi-structured data and match each line against the file-patterns.
    - 1.7. Rivet should produce a matching report that correctly represent pattern percentage in the sample data
    - 1.8. Rivet should provide user the ability to give name to newly generate pattern and RPL file if user did not use prune percentage option when running Rivet
    - 1.9. Rivet should output an error message and ask user to reselect patterns if user select invalid number of patterns from matching report

2. Non-Functional Requirements
    - 2.1. Rivet shall match the data against the generated file-patterns at least two times faster than it would against the brute force method.
    - 2.2. Rivet's output should be as accurate as the output generated from the brute force method.

3. Constraints
    3.1. Rivet shall use Rosie Pattern Language to generalize files into file-patterns.
    3.2. Rivet only works on Rosie version 0.99
    3.3. Rivet have to user python version 2.7.2 or higher
    3.4. Rivet only runs on the linux terminal

## Design

Author: Xiaoyu Chen
Editor: James Baggs

### System Flow

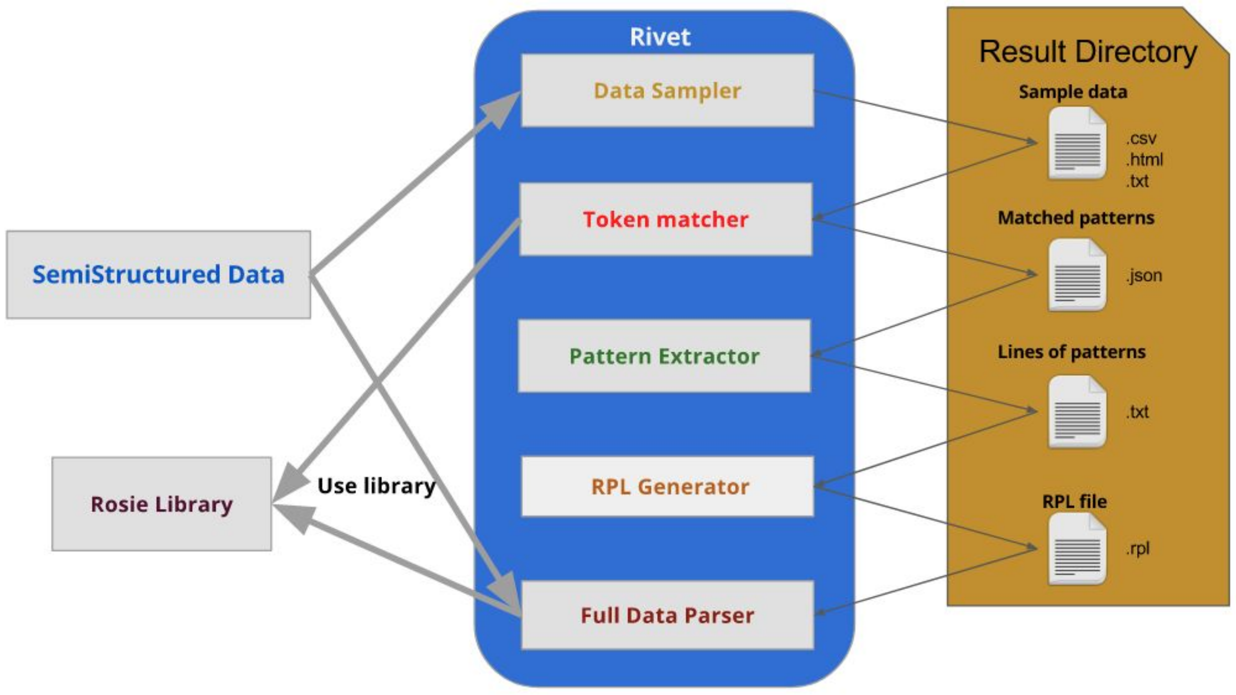


Figure 1 - System flow design

The architecture of Rivet's main flow is shown above in figure 1, and highlights both the modularity of the system, and the linear flow of data from its entry as semi-structured data to its exit as an RPL-encoded file pattern. It additionally showcases how the Rosie Library is utilized both in the early stage of the system system to generating our line-patterns and later how those same line-patterns are passed back to rosie to analyze the full semi-structured dataset.

This system's modular design, file outputs, and utilization of the Rosie library for token matching was motivated by several factors, and provides several benefits. Utilizing the Rosie library for token matching ensures that the system's generated RPL will match the naming convention and data types of the user's currently selected Rosie version. This means that when Rosie updates in the future, Rivet will continue to work on the system if Rosie's library calls remain the same, and it will require much less effort to update the Rivet system in the case that Rosie changes its syntax significantly. Making each section of the system modular makes unit testing and debugging easier, but we were also motivated when recognizing that several of Rivet's system modules are useful on their own outside of the system. The Pattern Extraction stage is highly dependent on the Token Matching stage, but the sampling implementation can be useful outside of the system for efficiently sampling datafile lines of any file, and the Full Data Parser can be useful for data analysts to test the match percent their custom-coded RPL file on a given dataset.

The flow of data in the Rivet system is as follows: First, the data enters the data sampler where the semi-structured data is expected to be in a format where one line is one data row. This is a reasonable expectation for formats such as csv or text, but other formats may need to be preprocessed by future developers to meet our system's expectation. The sampler randomly chooses lines from the semi-structured data to generate the sample dataset. The percentage of lines to be sampled is determined by the user. User is provided with a command line option to enter the sample size when running Rivet in the Terminal, the default of sample size is 100% if the sample size option was not provided. After the sampling process data sampler will output a sample data file that has the same file format as original data file. Then the brute force module will read in the sample data and using rosie library to identify the patterns within the lines. The brute force module outputs a json file that include all matched patterns and this json file will read in by Pattern Extractor to perform pattern extraction, it also calculate the frequency of each pattern appears in the sample data, and sort them in a decreasing order of frequency, it generate a text file with all patterns from sample data which can be read in by RPL generator. A match report will be generate by RPL Generator and provide user the ability to select multiple patterns to form a customized RPL file, if user provide a prune size when running Rivet, the RPL Generator will automatically select all patterns that are above this size to form a new RPL file. Then the newly generate RPL file  is used as input when processing the entire data file in the Full Data Parser.

## File-Patterns Data Structure

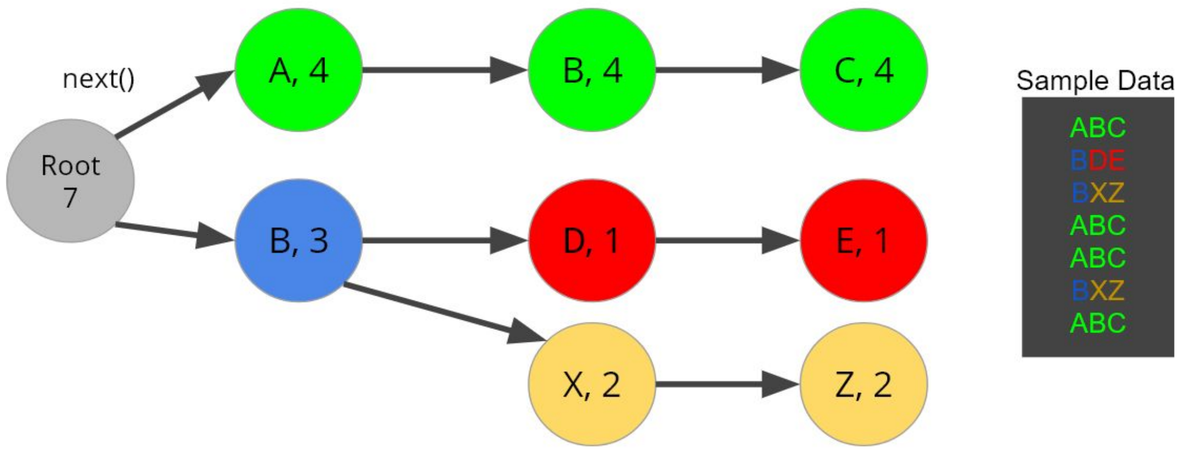
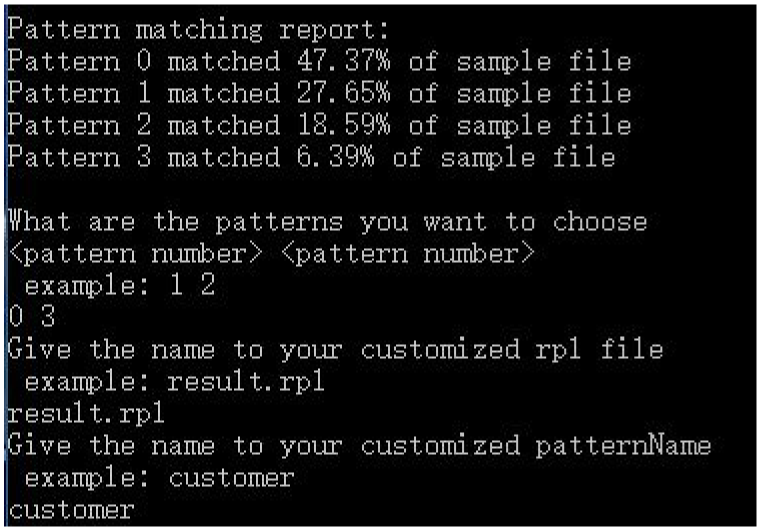


Figure 2 - File-patterns linked list tree data structure

The data structure of a file-pattern is implemented as a linked list tree, an example of which is shown in Figure 2. Each node in the linked list will contain the name of the pattern, a counter, and a list of pointers pointing to the next pattern(s) in the file-pattern(s). The nodes that the root node points to are the beginning of a line-pattern. For example, in the figure above, the root node points to the pattern B, and pattern B points to two other pattern nodes which means two file-patterns begin with pattern B. The root node does not contain a pattern name. It will keep a count of all existing file-patterns, while the other nodes keep a count of how many times they occur within file-patterns. The purpose of the counters is to determine the frequency of the file-patterns while minimizing the amount of memory required to store this information, and it has the added benefit of recognizing pattern prefixes. The most frequent pattern will be shown as the first pattern when generating pattern report.

## Pattern Match Report



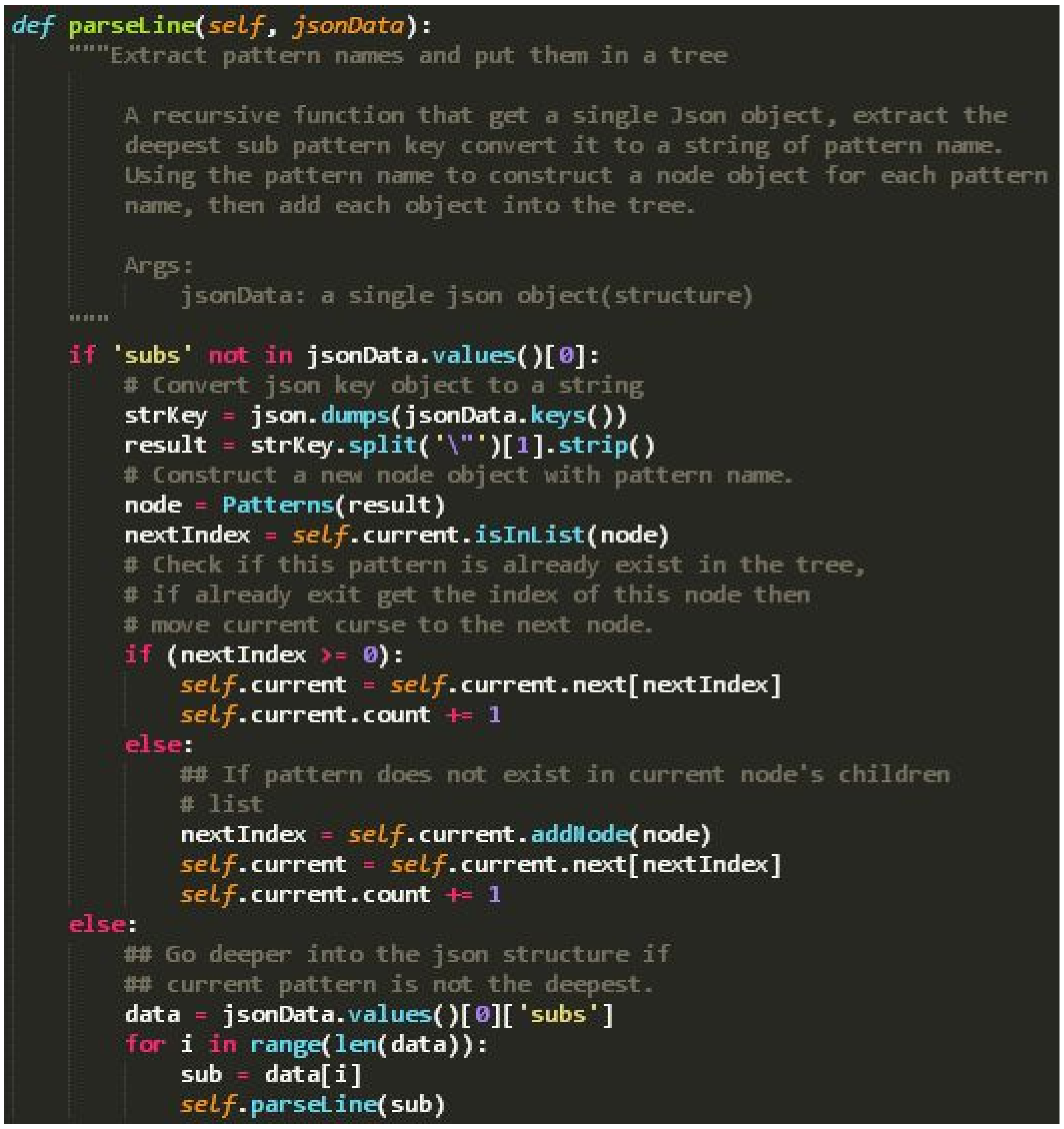
```
Pattern matching report:
Pattern 0 matched 47.37% of sample file
Pattern 1 matched 27.65% of sample file
Pattern 2 matched 18.59% of sample file
Pattern 3 matched 6.39% of sample file

What are the patterns you want to choose
<pattern number> <pattern number>
 example: 1 2
0 3
Give the name to your customized rpl file
 example: result.rpl
result.rpl
Give the name to your customized patternName
 example: customer
customer
```

Figure 3 - Pattern Match Report for customized pattern selection

An example of pattern match report is shown in Figure 3, the right side is an example of lines within the sample data we get from the Rivet data sampler, after all the process of brute force and pattern extraction user will get a report that is similar to the left side of Figure 3. As previously mentioned, the first line pattern is the most frequent one that appears in the sample data, which is 10 times and 50% of sample data in this example, and the pattern 2 is 25%, pattern 3 is 15% and the last one is only 5%. After the generation of this report, Rivet asks user to select one or more patterns to form the customized RPL file, in this example user may chose first 3 patterns and ignore the last one to generate a new RPL file, this functionality allows user to avoid potential corrupt data or error message within the data file, in this case is the last 5% pattern may match. On contrary, using the only the last pattern could help user quickly identify corrupt data and error message from the data file, after the selection of patterns user is asked to name their new pattern and a name for the new RPL file. If user provide a prune size when running Rivet, Rivet will automatically select all patterns that have a match rate above the provided prune size, then generates a RPL file and new pattern with default names.

## Low Level Design

Pattern Extraction

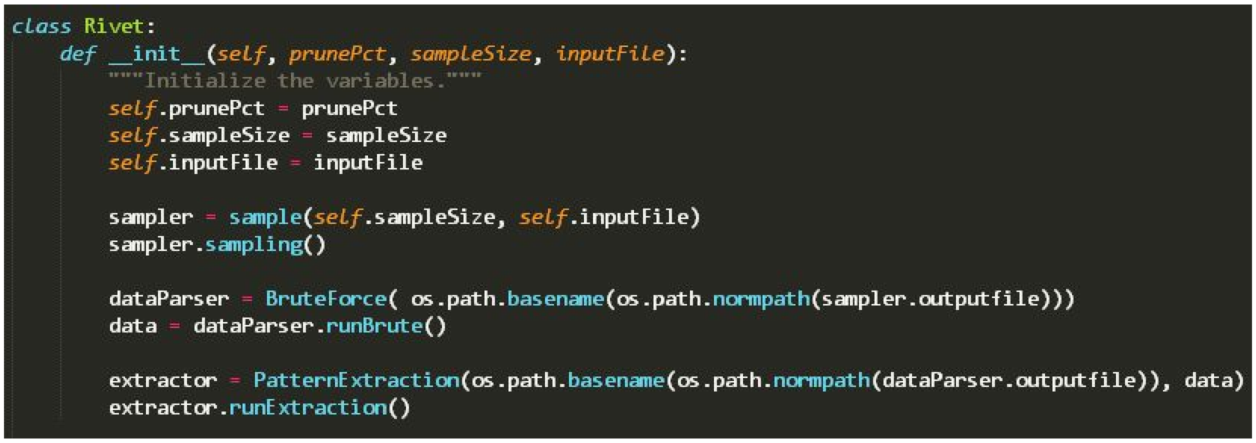


```
def parseLine(self, jsonData):
    """Extract pattern names and put them in a tree

        A recursive function that get a single Json object, extract the
        deepest sub pattern key convert it to a string of pattern name.
        Using the pattern name to construct a node object for each pattern
        name, then add each object into the tree.

        Args:
            jsonData: a single json object(structure)
    """
    if 'subs' not in jsonData.values()[0]:
        # Convert json key object to a string
        strKey = json.dumps(jsonData.keys())
        result = strKey.split('\"')[1].strip()
        # Construct a new node object with pattern name.
        node = Patterns(result)
        nextIndex = self.current.isInList(node)
        # Check if this pattern is already exist in the tree,
        # if already exit get the index of this node then
        # move current curse to the next node.
        if (nextIndex >= 0):
            self.current = self.current.next[nextIndex]
            self.current.count += 1
        else:
            ## If pattern does not exist in current node's children
            # list
            nextIndex = self.current.addNode(node)
            self.current = self.current.next[nextIndex]
            self.current.count += 1
    else:
        ## Go deeper into the json structure if
        ## current pattern is not the deepest.
        data = jsonData.values()[0]['subs']
        for i in range(len(data)):
            sub = data[i]
            self.parseLine(sub)
```

Figure 4 - Data Structure implementation

Above is the low level design for the data structure we implemented within the Rivet. The purpose of this function is to extract pattern token of each line from JSON file, this function is a recursive function that takes a JSON object and extracts the deepest pattern tokens which can be then used to check if it’s already exist in the tree, if the pattern already exist in the tree find the node’s index of that pattern and increment the counter by 1, if the pattern does not exist in the tree, create a new pattern node object then add the new pattern node as a child of previous pattern node.

Rivet Run All Modules


```
class Rivet:
    def __init__(self, prunePct, sampleSize, inputFile):
        """Initialize the variables."""
        self.prunePct = prunePct
        self.sampleSize = sampleSize
        self.inputFile = inputFile

        sampler = sample(self.sampleSize, self.inputFile)
        sampler.sampling()

        dataParser = BruteForce( os.path.basename(os.path.normpath(sampler.outputfile)))
        data = dataParser.runBrute()

        extractor = PatternExtraction(os.path.basename(os.path.normpath(dataParser.outputfile)), data)
        extractor.runExtraction()
```
Figure 5 - Rivet combine all module together

This is where Rivet start its process of RPL generation, it combines all modules together to do the job, the first modules need to run is the sampler, that will sample the original data into a small portion of dat, then send that data into dataParser and use rosie to get all patterns in the sample data and generate a JSON file that can be used by PatternExtractor. PatternExtractor will extract all patterns and generates matching report to user to select the final pattern

## Implementation

Author(s): James Baggs

Editor: Yuxu Yang

### Iteration 1

Our first iteration focused on producing JSON output of data extracted by Rosie from an input file. The purpose of this iteration was to provide our team with experience in Python and Rosie before continuing on to more complex portions of the project, but the brute force code ended up being a crucial portion of our final project. The BruteForce.py file remains similar to its initial implementation, and is used to produce the sample line-pattern extraction from Rosie which passes data to the PatternExtractor class.

This implementation did not directly satisfy any requirements on its own, though it provides a useful metric in terms of match speed and percentage through Rosie's python library that we later use to ensure that non-functional requirements 2.1 and 2.2 are satisfied. It is important that we later compared our extraction speeds to BruteForce.py's time on a full dataset because the python library runs much slower than Rosie's command line interface, and we would otherwise have an inaccurate comparison.

### Iteration 2

Iteration 2 significantly took steps toward the generation of a RPL file-pattern based off of an analyzed sample of semistructured data. Sample.py was created to randomly sample lines from a file to be passed to BruteForce.py, and Patterns.py & PatternExtraction.py were created to extract the line-patterns from the JSON output produced by BruteForce. This was largely successful, but we caught one particular error which we named the "leftover problem". This problem was caused by slight formatting differences in rosie's JSON output and our RPL generation, that would cause newline characters to not be recognized, and also caused sub-pattern prefixes such as AB to match a line containing ABC and it would throw away the C from extracted data. This is a problem that we began addressing in this iteration and found a complete solution to in Iteration 3. We additionally generated data, which we labeled "UniformGeneratedData" that artificially provided a diverse set of datatypes in a uniform format of only 1 unique line-pattern per file for simplicity.

This implementation satisfied the requirement 1.3 and the first part of requirement 1.4, which is that it generates file-patterns internally, though the requirement 1.6 is only completed in iteration 3 when these file-patterns are printed in RPL format for rosie to use. Using UniformGeneratedData in a csv format as a source for patterns to be extracted from satisfied the requirement 1.2.1 which specifies that CSV files should be accepted as input.

<u>Iteration 3</u>

Iteration 3 mainly focus on the fourth module of Rivet system, the RPL generator, in order to generate the RPL for user, this process is implemented in the FinalRecognition.py which read in a text file that contains all patterns within the sample data, and generates the pattern report to the user. We also implement command line argument options for user to chose sample size and prune percentage, if the user do not provide a desire prune size, after generating the matching report user will be asked to select one or more desired patterns to form the customized pattern, after the selection of patterns, user need to provide name for both the customized pattern and the rpl file. On the contrary if the user provide a prune percentage when running Rivet, the system will automatically generate customized pattern which combine all patterns that have a matching percentage above the provided prune percentage. Rivet will automatically give name for both new RPL file and the new pattern. The newly generate RPL file will be stored into the result directory in the Rivet system, and user is able to use data parser to load the RPL file into rosie and parse original data.

This iteration satisfied the rest of requirements, which is to use newly generated RPL file to parse the original data, and any data that has the same pattern which is include in the customized pattern should be matched by rosie. Data parser will output all data that is matched by rosie into a json file, and user is able to check the data within the json file.

Optimize sample.py

<u>Iteration 4</u>

Iteration 4 concluded the Rivet system project requirements by providing a more robust test suite to prove that our system was working as intended and additions to the user interface such as progress bars, error handling, and pattern customization. testExtractionTime.py measures that a specified RPL file extracts data from a file more efficiently than Rosie's brute force. It provides a user with an option for the number of times that they want to test the data parsing, in case of large computational time variance, such that a statistical significance analysis could prove the time to be significantly better. In FinalDataPattern.py, we test the match percentage of the RPL file against the full dataset (as opposed to the sample dataset match percentage in FinalRecognition.py) and print that to the user for black box testing.

Additionally we added a tool named PruneSampleAnalysis.py that provides users with the ability to analyze a directory of files and find what variables might influence the match% and speed of a file. This program runs the Rivet system multiple times with different sample sizes and prune percentages and graphically prints to the user what the corresponding match percentages for the file are.

We additionally optimized some overhead computation that our system was performing in the pattern extraction (Many read/writes to file), and found additional data sources in different formats (such as HTML, TSV, and TXT) to attempt our pattern generation and pattern extraction against.

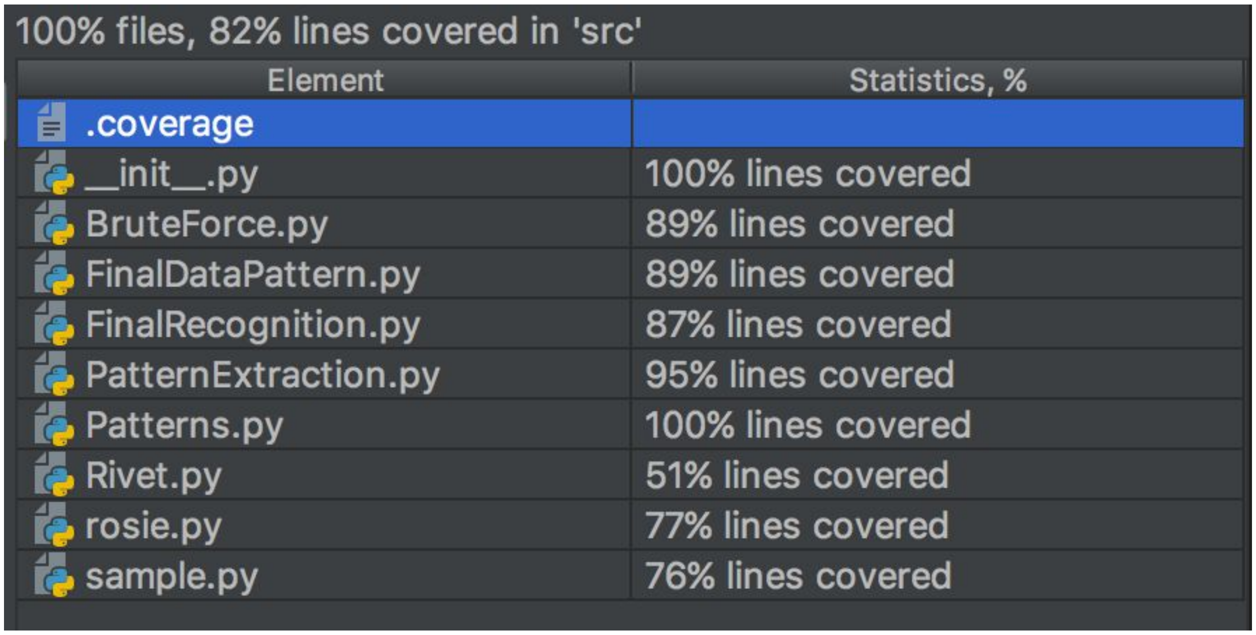
This suite of testing confirms that the non-functional requirements 2.1 and 2.2 (speed and match%) pass properly. What we found is that the more uniform the data, the more quickly data would be parsed using our system-generated RPL file. The use and confirmation of HTML, TSV, and TXT files also satisfy our requirements in 1.2 which specify that the system should handle many data types. However, what we found when we attempted to use organic data from data.gove is that many data sources did not work for various reasons. We believe some of these reasons have to do with Rosie v0.99k's pattern matching module, but we also know that our algorithm is not robust enough to handle extremely non-uniform data. For example, the provided datafile in our github named NC Zip Codes.csv has dozens of "features" (comma separated columns) and many missing data points. This leads the ~5,000 line file to have roughly ~5,000 unique line-patterns. Some potential solutions to these types of problems are ongoing sources of research in the data mining field and, with respect to the Rosie/Rivet system, we have suggested potential solutions in the "Work for future teams" section.

## Test Plan and Results

Author(s): James Baggs
Edited by: Yuxu Yang

The team has selected PyUnit as the project's unit testing tool while Coverage.py will be used as our coverage tools. Our black box test suite verifies that the high-level functional requirements of the system are satisfied, using both manual observations and automated black box testing scripts. For example, the black box testID named file PatternParseSpeed uses the provided ExtractionTime.py script to verify that a Rivet-generated file pattern results in data extraction speed increases over Rosie's brute force. These tests and tools focus on the three main functional requirements – speed, accuracy, and sampling – all have specific tools for testing purposes, and all require specific criteria for a passing test case. The parsing speed test requires that a uniform data file is able to be parsed at least twice as quickly with a file pattern created by our system than by Rosie's Brute force approach. To facilitate the timing of this test, we have suggested that testers use /usr/bin/time, but we have also created a test script which automates this process, named ExtractionTime.py. Similarly, our integrity test will eventually include a python script to compare the types of matches of Rosie's brute force extraction to Rosie's extraction of data given a generated file pattern.

Our unit testing covered all the source programs, excluding black box testing/utility programs: ExtractionTime.py and PruneSampleAnalysis.py. Due to our project design, our program will have one data structure and multiple data out file. The methods are used in the unit testing is covering how the program react to the certain choices based on our example data. The result from our program will match certain number which calculated by us. The result of our unit test is shown below. We achieved 82% of line coverage.



100% files, 82% lines covered in 'src'

| Element | Statistics, % |
|---|---|
| .coverage | |
| __init__.py | 100% lines covered |
| BruteForce.py | 89% lines covered |
| FinalDataPattern.py | 89% lines covered |
| FinalRecognition.py | 87% lines covered |
| PatternExtraction.py | 95% lines covered |
| Patterns.py | 100% lines covered |
| Rivet.py | 51% lines covered |
| rosie.py | 77% lines covered |
| sample.py | 76% lines covered |

## Black Box Test Plan

For the black box test, we are using a testing data set name NewDataSet.csv in the Resources directory, and most of the tests will use this data file as input into Rivet system.

Our program currently passes all the black box tests we have included in this report. It is able to generate a valid report of matching patterns to the user, A user is able to chose different sample size to increase accuracy of the match result of the original data. When user do not provide a prune percentage, Rivet will provide the ability for user to select multiple patterns to form the final pattern of RPL file otherwise, if user input a desired prune percentage Rivet will automatically select patterns that are above the prune percentage and generate a RPL file with auto generate name.

| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
|---|---|---|---|
| progressBarAppears | Steps:<br>1) Run *$python Rivet.py -s 10 -p 1 NewDataSet.csv* | Total three Progress Bars should appears for sampling, parsing | Total three progress bars appears for sampling, parsing data, and pattern |

| | | | |
|---|---|---|---|
| | | data, and pattern extraction | extraction |
| generatingMatchResult | Steps:<br>2) Run *$python Rivet.py -s 10 -p 1 NewDataSet.csv* | A matching report should appears after pattern extraction for NewDataSet.csv there should be 4 patterns in the report | A matching report appears after pattern extraction with result of 4 patterns |
| sampleSize | Steps:<br>3) Run *$python Rivet.py -s 10 -p 1 NewDataSet.csv or Run $python Rivet.py --sampleSize=10 NewDataSet.csv*<br>4) Check the sampling result | The sample size should be 10% of the original data. For NewDataSet.csv the new sample size should be 1237 | The original data lines is 12376<br>The sample data set is 1237 lines |
| pruneSizeWithAutoGenerateRPL | Steps:<br>1) Run *$python Rivet.py -p 20 NewDataSet.csv or Run $python Rivet.py --prunePct=20 NewDataSet.csv*<br>2) Check the generated RPL file | A new RPL file named auto.rpl in the result directory, a pattern name auto should be in the auto.rpl contains only pattern that matched 20% of sample data | A new file name auto.rpl appears in the result directory, a new pattern name auto contains only 2 patterns one is 45.92% another one is 29.83% matched sample data. |
| customizedPatternCreation | Steps:<br>1) Run *$python Rivet.py -s 10 NewDataSet.csv or Run $python Rivet.py --sampleSize=10 NewDataSet.csv*<br>2) Type in “1 2” hit enter<br>3) Type “result.rpl” hit enter<br>4) Type “customer” hit enter | A new rpl file named “result.rpl” should appears in the result directory, a new pattern named customer contains only pattern 1 and 2 from the matching report. | A new rpl file named “result.rpl” appears in the result directory, a new pattern named customer contains only pattern 1 and 2 from the matching report. |

| | | | |
|---|---|---|---|
| | 5) Check result directory<br>6) Open the result.rpl with a text editor | | |
| invalidPatternSelection | Steps:<br>1) Run *$python Rivet.py -s 10 NewDataSet.csv*<br>*2) Type “12” or “ ” or “-1” then hit enter* | A error message should appears says “Invalid index” and the program should ask user to select pattern again | A error message appears says “Invalid index” and the program asks user to select pattern again |
| filePatternParseSpeed | Preconditions:<br>● The file pattern named result.rpl with pattern “customer” should already be completed for the given sample data.<br>Steps:<br>1) Run command: python ExtractionTime.py -t 1 NewDataSet.csv result.rpl customer<br><br>2) Record the time for both time for both Brute Force and RPL time. | The time that it takes Rosie to parse data with a file-pattern should be less than or equal to half the length of time it takes to parse data by brute force. | The time that it takes Rosie to parse data with a file-pattern is 12.04 seconds which is less than or equal to half the length of time it takes to parse data by brute force 26.46 seconds |
| bruteForceJsonToFile | Preconditions:<br>Steps:<br>1) Run *$python Rivet.py -p 1 NewDataSet.csv*<br>*2) Wait program to finish and check result directory* | A file should be created which contains JSON output that is the same as the standard output Rosie’s default JSON encoded parsing of a given sample file (ie $*rosie -encode json basic.matchall sampledata.csv*) | Output.json is created and contains all lines of JSON data when NewDataSet.csv was used. The results appear to be the same as the output of rosie against the same file. |

| | | | |
|---|---|---|---|
| testModuleSampler | Preconditions: NewDataSet.csv is in the resources directory<br>Steps:<br>1) Run $python Sample.py 10 ../Resources/NewDataSet.csv | A new sample data file should appears in the resources directory with 1237 lines | A new sample data file appears in the resources directory with 1237 lines |
| allModulesOutputFiles | Steps:<br>1) Run $python Rivet.py -s 10 -p 1 NewDataSet.csv | There should be total 3 files appears in the result directory “auto.rpl, output.json, result.txt”, there should be one file appear in the resources directory : “sample_NewDataSet.csv” | 3 new files appear in the result directory: “auto.rpl, output.json, result.txt” 1 file appears in the resources directory: “sample_NewDataSet.csv” |
| testFinalDataParser | Preconditions:<br>● auto.rpl been generated for NewDataSet.csv<br>Steps:<br>1) Run $python FinalDataPattern.py NewDataSet.csv auto.rpl auto | Program should generate a match rate result, and output a json file named “customizedResult.json” in the result directory | Program generates a 100% match rate result, and output a json file named “customizedResult.json” in the result directory |

**Team Contact Information**

James Baggs - Team Leader, Support Manager, Coordinator of Documentation
jhbaggs@ncsu.edu

Xiaoyu Chen - Quality/Process Manager, Coordinator of Implementation
xchen27@ncsu.edu

Yuxu Yang - Development Manager, Coordinator of Design and Requirements Analysis
yyang21@ncsu.edu

## Suggestions for Future Teams

Author(s): James Baggs

We have discussed with our sponsor where our contribution to the Rosie project fits in the scope of data mining in general, and where future teams might work to improve the system further. Among these discussions was a description of why storing parsed data to an Elasticsearch/NoSQL database was important:

While our system will be able to leverage Rosie to automatically create file patterns and parse data more efficiently, this parsing of data is limited to matching expressions defined in the rosie language, or additional expressions which have to be manually coded by hand. A source for additional research and work is finding a way to generate these *expressions* automatically. That is, instead of having to manually code that the *time* pattern is composed of *hours, minutes and seconds*, create a system which is able to recognize that the *hours, minutes, and seconds* expressions are frequently co-located in a wide variety of data sources, and suggest to a user to name and create a new expression composed of these. Though Rivet's main stated purpose has been to improve the speed at which data is parsed, it achieves this by analyzing the frequency of patterns in a given file. By providing a method for our system to save the pattern frequencies to a NoSQL database instead of simply saving the parsed data, future teams can query this database with Elasticsearch and use data to analyze which expressions are frequently co-located, or in a particular order. These expressions are not limited to the expressions pre-defined in Rosie, and can also include fine-grained character expressions such as punctuation, special characters, and digits.