Multiple-Drug Interaction Analytics Platform

Developing a Minimum Viable Web Application for Visualizing Multi-Drug Interactions

Submitted: March 22, 2018

A Major Qualifying Project to be submitted to the faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science

By
Brian McCarthy
Andrew Schade
Huy Tran
Brian Zylich

In Collaboration With
Tabassum Kakar, *PhD Student*
Xiao Qin, *PhD Student*

Submitted To:

_____

Prof. Elke A. Rundensteiner
Worcester Polytechnic Institute

Abstract

Hundreds of thousands of patients report unwanted symptoms from using one or more drugs each year. It is virtually impossible to test every drug combination before a drug reaches the market. In this project, we reimplement and integrate two previously proposed tools to mine and visualize interesting pairs of drugs from report data, focusing on creating a single web application that is functional and maintainable. Furthermore, we research and implement new features including drug and reaction name standardization. Upon finishing development, we conduct a user study and technical evaluation to ensure our system meets the requirements and improves usability. Based on the results of our studies, we improve the usability and identify areas that require further attention.

Acknowledgements

This project would not have been possible without the help of our faculty advisor, Prof. Elke Rundensteiner, and our graduate student mentors, Tabassum Kakar and Xiao Qin. They provided extensive guidance and feedback throughout our development efforts. Additionally, we would like to thank Prof. Lane Harrison for his guidance, especially with the data visualization and user evaluation aspects of the project. We are thankful to Worcester Polytechnic Institute and the Computer Science department for providing us with the opportunity to work on this project.

We would also like to thank Ermal Toto and the WPI ARC for providing the computational resources to host our application on the web, facilitating testing and evaluation.

Finally, we would like to thank the FDA for their assistance in establishing requirements for our project and providing some (redacted) data essential for demonstrating the capabilities of our project.

Executive Summary

Every year, more than 100,000 deaths and 2 million hospitalizations are caused by the side effects of prescription drugs or the unintended interactions from taking multiple drugs simultaneously. These adverse drug reactions (ADRs) are one of the leading causes of death worldwide. While extensive clinical trials are conducted to identify the ADRs associated with a single drug before that drug reaches the market, it is virtually impossible to test how a drug will interact with every other drug or combination of drugs. Therefore, multi-drug adverse reactions (MDARs) are identified primarily after the drug reaches the market. Healthcare professionals and consumers are encouraged to submit reports through the FDA's Adverse Event Reporting System (FAERS) in the event that they witness or experience what they believe may be an ADR. Safety evaluators with the FDA then analyze these reports to identify groups of drugs that cause an adverse reaction when taken together. To assist FDA safety evaluators with their goal of identifying MDARs, Xiao Qin and Tabassum Kakar developed the Multi-Drug Adverse Reaction Analytics Strategy (MARAS) system and also designed a visual paradigm, which they called Drug-Drug Interactions via Visual Analysis (DIVA). These systems harness association rule mining to identify potential MDARs from FAERS reports and provide an interactive visualization of the 2-drug rules mined by MARAS, respectively.

The goal of our project was to develop a minimum viable product to demonstrate how Kakar and Qin's work could be used to assist FDA safety evaluators. To do this, we unified the MARAS and DIVA systems into a single application, the Multi-drug Interaction Analytics Platform (MIAP). While developing MIAP, we completely reimplemented the MARAS and DIVA systems to facilitate their unification into one system and their future maintainability. Throughout the process, we improved the MARAS technology by adding smarter drug name and adverse reaction processing. We also improved the DIVA system by increasing visual appeal and overall usability. Finally, we conducted a usability evaluation to test how MIAP matched up against the requirements specified by Kakar and Qin's cooperative research with the FDA and highlight areas that could use improvement.

Currently, FDA safety analysts are confronted with the challenge of sifting through millions of new reports each year in search of MDARs. For now, these personnel are approaching this task without the support of software for mining potential interactions, visualization of these interactions, or the screening of potential interactions based on confidence scores or association with severe adverse reactions. This means that safety analysts have to spend time scanning through reports before prioritizing potential drug combinations for further investigation. Then, when they have prioritized the drug combinations of interest, they must retrieve all reports related to each drug combination before applying domain knowledge to determine how to proceed with a clinical investigation or potential drug regulation.

With the assistance of MIAP, the prioritization of potential MDARs and subsequent analysis of those MDARs is simplified. Filters can be applied to the visualization to look

at the MDARs with the highest confidence scores. These drugs can then be sorted by the severity of the associated adverse reactions. Then, all of the reports associated with the drug or hypothesized MDAR can be retrieved with the click of a button.

While much of the functionality of MIAP was previously available in the MARAS and DIVA systems, there are several notable improvements. The previous MARAS system was composed of multiple programming languages, lacked documentation, and did not take full advantage of Object-Oriented programming concepts. Additionally, MARAS did not perform any cleaning on drug names or adverse reaction names, and did not label mined rules as known or unknown. Meanwhile, the previous DIVA system also lacked sufficient documentation and was completely separate from the MARAS system. Thus it required manual intervention to move files between the two systems.

As a result of this project, the new MARAS system was reimplemented using just one programming language and taking advantage of encapsulation and object generation. Additionally, javadoc-style documentation was used to provide internal and external guidance on the use of the system. MARAS now performs cleaning on drug names and adverse reactions to assist in the rule mining and known-rule labeling processes. The DIVA system also underwent a complete reimplementation, separating each feature into its own component file and adding javadoc-style comments for future developers. Finally, MIAP unifies MARAS and DIVA, meaning that no manual intervention is required to go between the user interface for visualization and the rule mining process, and vice versa.

With the addition of these new features, users can now upload FAERS data via the user interface, automatically starting the MARAS analysis on the new data combined with all previously uploaded data. Then, when the MARAS analysis finishes, the users' visualization will be updated automatically. With improved usability aspects, it will be easier than ever to prioritize and analyze MDARs. Lastly, with refactored code and extensive documentation, future developers will be able to more easily contribute to this project. This project will serve as a better model should the FDA choose to adopt a similar application in support of their drug safety investigation.

Upon completion of this project, we have developed a list of recommendations for improvements that can be made to MIAP in the future. These recommendations are as follows:

1. Add a database to store reports and rules in order to decrease the time of retrieving information about specific reports or drugs.
2. Improve the usability of the Reports View and add annotation functionality to allow FDA safety analysts the ability to give domain insight directly within our tool.
3. Allow for the visualization of custom datasets or a subset of the existing dataset, as we currently have chosen to support only appending new data from FAERS into our tool.
4. Conduct an evaluation with FDA personnel to ensure MIAP meets their needs,

receive additional suggestions, and determine whether they agree with the metaphors currently employed in MIAP.

Contents

List of Tables

List of Figures

1. Introduction

More than 100,000 deaths and 2 million hospitalizations, annually, are caused by the side effects of prescribed drugs or the unintended interactions from taking several drugs at once ("Preventable Adverse Drug Reactions: A Focus on Drug Interactions" 2016, placeholder). These side effects, called Adverse Drug Reactions (ADRs), create a public health problem that makes it one of the leading causes of death worldwide. While the ADRs directly caused by a single drug are well-researched, often documented well before drugs reach the public market, there is much less documentation about how drugs will act in combination with each other. Polypharmacy, or having multiple drugs prescribed at once, affects more than 40% of elderly people living at home and some of these patients regularly take up to 18 different drugs. The number of combinations of drugs that are taken together is far larger than can be researched in depth by medical professionals, so it is necessary to have tools that can use patient data, collected after the drugs have gone to market, to assist researchers in identifying probable candidates for multi-drug interactions.

Originally developed by Xiao Qin and Tabassum Kakar (Kakar 2016, Kakar and Qin (2017–2018)), the Multi-Drug Adverse Reaction Analytics Strategy (MARAS) and Drug-Drug Interactions via Visual Analysis (DIVA) systems are designed to aid analysts in discovering these multi-drug interactions. MARAS focuses on mining FAERS, a dataset provided by the FDA that contains ADR reports submitted around the globe, to find groups of drugs likely to be contributing to Multi-drug Adverse Reactions (MDARs), sorted by a severity score. DIVA, on the other hand, focuses on visualizing the 2-drug interactions found by MARAS. It allows users to see the network of interactions at a glance and find the raw FAERs reports contributing to the groupings found by MARAS.

For these tools to be used by professional analysts, they need to be tightly integrated, efficient, and stable. Our project evaluated the MARAS and DIVA systems as they were previously, implemented improvements in code quality and performance, and integrated the systems into a fully-functional web application that can be used by analysts to discover new MDARs from FAERS data. We focused on the usability and maintainability of the system, while also developing additional features and designs, to ensure that FDA employees and other analysts can utilize the application as effectively as possible. The DIVA software can improve the speed, efficiency, and accuracy of MDAR discovery and allow the FDA to more efficiently identify and publicize these dangerous interactions.

1.1. Motivation

When patients experience unwanted symptoms while taking a drug, they submit reports, called Adverse Drug Events (ADEs), to an FDA-run database, the "FDA's Adverse Event Reporting System" (FAERS) ("Questions and Answers on Fda's Adverse Event Reporting System (Faers)" 2017). As more and more people are being prescribed multiple drugs, the number of ADEs sent to FAERS is increasing rapidly. This database receives millions of new records every year, which makes it impossible for individual analysts to take full advantage of the amount of data being collected. There is need for a system that was able to use the wealth of data in the FAERS database to allow drug evaluators make informed decisions about where to focus more in-depth investigations.

This niche requires an application that can provide evaluators an at-a-glance overview of the possible drug interactions indicated by the data, as well as a system for understanding which associations are the most common or most severe in order to inform investigator priorities.

1.2. Previous Research

Analyzing the multitude of drug interactions requires both a back-end mining algorithm that can report on the most common or likely combinations of drugs and an interface that can allow evaluators to quickly understand those interactions and then dive deeper into individual reports to study further. Our graduate student partners (Kakar 2016, Kakar and Qin (2017–2018)), researched two techniques that address the back-end space and interface respectively. Kakar and Qin created the Multi-drug Adverse Reactions Analytic System (MARAS), which ranks mined associations by their 'contrast score' a metric that quantifies how closely an ADR is associated with a pair of drugs rather than each drug individually.

They also developed the Drug-Drug Interactions via Visual Analytics (DIVA) paradigm, which is a visualization, with several different views to allow the interactions—found and ranked in MARAS—to be analyzed in different contexts within the same application. Unfortunately, these two projects were conducted largely independently. Thus, it was a labor-intensive, manual process to take the associations ranked in MARAS and integrate them into the DIVA visualization.

1.3. Contributions

Our project focused on developing a **minimum viable product** to demonstrate how Kakar and Qin's work could be used to assist evaluators discovering and testing multiple-drug adverse reactions. We unified MARAS and DIVA into a single application, the Multiple-drug Interaction Analytics Platform (MIAP), which focused on creating a

smoothly interactive user experience. Our results, supported by an evaluation, demonstrate the utility of the technology for supporting the drug evaluation workflow.

As a minimum viable product, our project demonstrated the capability to:

- Correctly intake, mine, and rank new FAERS data provided by a user with no further human intervention,
- View all ranked hypothetical drug-drug interactions at-a-glance,
- Differentiate between unknown hypothetical drug-drug interactions and known interactions that have already been researched,
- Focus on and select hypothetical drug-drug interactions for further study,
- Prioritize focused hypothetical drug-drug interactions by various criteria,
- Facilitate detection of severe Adverse Drug Reactions, and
- Link to the underlying FAERS reports.

Future work can conduct experiments on the customer value of MIAP, determining how valuable a tool like it would be to evaluators. If the system is found to be worth developing further, additional improvements should me made to expand MIAP into an application that fulfills all features required of a software suite that can facilitate the complete evaluation process. This requires research in determining exactly what other workflows in the MDAR evaluation process should be integrated into a single application.

2. Background

The Food and Drug Administration (FDA) is responsible for approving drugs for medical use and for keeping up-to-date on the risks associated with taking those drugs. The Center for Drug Evaluation and Research (CDER) is the specific department at the FDA that is responsible for ensuring that "safe and effective drugs are available to ensure the health of people in the United States" ("About the Center for Drug Evaluation and Research" 2018). This department regulates both over the counter and prescription drugs, from fluoride toothpaste and sunscreen to narcotic painkillers and cancer medications.

One part of this is ensuring that drugs continue to stand up to safety measures after they've gone to market. Clinical trials cannot explore all the ways that using a drug might be harmful, so the FDA must rely on data collected from patients and hospitals during the course of their usage. One tool that the FDA uses is by collecting reports on Adverse Drug Reactions.

An Adverse Drug Reaction (ADR), or adverse event, is defined by the Food and Drug Administration (FDA) as an "untoward medical occurrence associated with the use of a drug". Every year, there are over 2 Million serious ADRs, resulting in over 100,000 deaths("Preventable Adverse Drug Reactions: A Focus on Drug Interactions" 2016). ADRs are the fourth leading cause of death, surpassing automobile deaths, diabetes, and pulmonary disease. In addition, over 350,000 ADRs occur in nursing homes annually. To gather information on ADRs, the FDA uses FDA's Adverse Event Reporting System (FAERS)("Questions and Answers on Fda's Adverse Event Reporting System (Faers)" 2017). The FAERS database contains information on ADR reports submitted to the FDA by healthcare professionals and consumers. With the mass of ADRs reported through FAERS the lack of automated tools to perform preliminary analyses limits the usefulness of the data collected.

An interaction between multiple drugs causing one or more ADR(s) (such as taking Aspiring and Warfarin together causing bleeding) is known as a Multi-Drug Adverse Reaction, or MDAR. One of the more difficult analyses to extract from these data is multiple-drug interactions. Up to 40% of elderly people are prescribed more than six medications simultaneously, in so many different combinations that testing them all, individually, for side effects that are derived from the interaction between two or more of those drugs impossible.

In order to create an automated screening process for which sets of drugs are candidates for causing MDARs, Kakar (2016) developed the Multi-drug Adverse Reactions Analytic System (MARAS).

2.1. MARAS

MARAS utilizes Association Rule Learning to identify ADRs which are most significant. There are several challenges associated with using Association Rule Learning with ADRs. Applying Association Rule Learning to a large set of drugs and ADRs, such as the set generated by FAERS, results in an excessively large number of associations. It is difficult to reduce the number of rules to a manageable scale while still guaranteeing that potentially important rules are not missed. In addition, the basic association measures used with Association Rule Learning, such as support and confidence, do not appropriately rank drug-ADR associations, because they only consider individual association rules, while MARAS has to consider the correlation among different rules. The MARAS methodology attempts to address these concerns to sufficiently filter and rank drug-ADR associations in a meaningful way.

2.1.1. Association Rule Learning

Association Rule Learning refers to the process of creating Association Rules from a set of transactions, where an Association Rule is defined as a description of the probability that, given the occurrence of one event, another event will occur. In other words, "it helps find the relationship between objects that are frequently used together" (Gollapudi 2016). One common example of an association rule would be that if a customer purchases an iPad, they are likely to also purchase an iPad case.

The **support** of an association rule is the frequency with which the items appear together compared to the total number of transactions. More formally, the support of a rule $X \rightarrow Y$ is defined as: $Support = frq(X, Y)/N$ where N is the number of transactions.

The **confidence** of an association rule $X \rightarrow Y$ is the frequency with which the items appear together compared to the frequency that X appears by itself; in other words:

$$Confidence = \frac{frq(X, Y)}{frq(X)}$$

The **lift** of an association rule $X \rightarrow Y$ is the support of the rule compared to the product of the supports of X and Y individually.

$$Lift = \frac{support(X, Y)}{support(X) \times support(Y)}$$

2.1.2. Important Concepts for MARAS

A Drug-ADR association R is a pairing of a set of Drugs D and a set of ADRs A. The association is considered **explicitly supported** if at least one report exists that refers

exactly to the Drugs and ADRs in the association, with no others. Meanwhile, the association is considered **implicitly supported** if it exists as partial interpretations of two or more reports. For example, consider a report R_1 with a set of drugs $D_1 = \{d_1, d_2, d_3\}$ and a set of ADRs $A_1 = \{a_1, a_2\}$, and a report R_2 with a set of Drugs $D_2 = \{d_1, d_2, d_4\}$ and a set of ADRs $A_2 = \{a_1, a_2, a_3\}$. In this case, the association $d_1, d_2, d_3 \rightarrow a_1, a_2$ is explicitly supported, since R_1 refers exactly to this association. The association $d_1, d_2 \rightarrow a_1, a_2$ is implicitly supported, because it is a partial interpretation of both R_1 and R_2. An association is a **spurious association** if it is neither explicitly nor implicitly supported.

An association is considered **closed** if it does not contain partial information of another association. Consider $R_1 = d_1, d_2, d_3 \rightarrow a_1, a_2$ and $R_2 = d_1, d_2 \rightarrow a_1, a_2$. In this case, R_2 is *not* closed, because it presents partial information of R_1. In other words, R_1 provides richer information than R_2.

Consider a report R_1, an association of a set of drugs D_1 and adverse reactions A_1; similarly, R_2 is an association of a set of drugs D_2 and adverse reactions A_2. R_1 is a **contextual association** of R_2 if D_1 is a subset of D_2 and A_1 is a subset of A_2. For example, let $R_1 = d_1, d_2, d_3, d_4 \rightarrow a_1, a_2, a_3, a_4$; let $R_2 = d_1, d_2 \rightarrow a_1, a_3$. In this case, R_1 is a contextual association of R_2. A **contextual association cluster (CAC)** contains an explicitly or implicitly supported association, and its contextual associations.

$\mathcal{R}$	[Furosemide] [Isosorbide] [Aspirin] ⇒ [Myocardial Infarction]
$\tilde{\mathcal{R}}^2$	$\tilde{\mathcal{R}}^2_1 \equiv$ [Furosemide] [Isosorbide] ⇒ [Myocardial Infarction]
	$\tilde{\mathcal{R}}^2_2 \equiv$ [Furosemide] [Aspirin] ⇒ [Myocardial Infarction]
	$\tilde{\mathcal{R}}^2_3 \equiv$ [Isosorbide] [Aspirin] ⇒ [Myocardial Infarction]
$\tilde{\mathcal{R}}^1$	$\tilde{\mathcal{R}}^1_1 \equiv$ [Furosemide] ⇒ [Myocardial Infarction]
	$\tilde{\mathcal{R}}^1_2 \equiv$ [Isosorbide] ⇒ [Myocardial Infarction]
	$\tilde{\mathcal{R}}^1_3 \equiv$ [Aspirin] ⇒ [Myocardial Infarction]

Figure 2.1.: CAC associations

There are two factors to consider when measuring if a Drug-ADR association indicates a severe Multi-Drug Association Rule (MDAR): How strong the association is between the ADRs and drugs, and how strong the association is between the ADRs and each individual drug, or a subset of the drugs. For the former, we can use the traditional **confidence** measure, referenced above. For the latter, we can use the **contrast** measure. A high contrast indicates that the target association has a high confidence, and all of the contextual associations within its CAC have low confidence.

2.1.3. MARAS System

The MARAS System methodology describes how the Association Rule Learning methods detailed above can be used to present Adverse Drug Reaction information in a descriptive

and informative manner. The first step in this process is the Data Processor, which extracts and cleans drug and ADR data from an ADR report (such as a FAERS report) so it is in the appropriate format to be used by the MDAR signaler; The Data Processor also removes any duplicate reports. Next, the MDAR Signaler analyzes the output of the Data Processor to identify the non-spurious Drug-ADR associations along with their Contextual Associations, then computes the contrast score of each CAC to identify the strength of the signal.

The MARAS code received at the beginning of this project consisted of two main parts. The first part of the system focused on using association learning to mine hypothetical drug-drug interactions from the FAERS data and the other part was responsible for processing and filtering the data for the data mining part and for having usable output. The processing part was further divided into three sections: preprocessing, closure generation, and result output.

The Preprocessor builds a copy of the initial data-tables pulled from the FAERS database by generating unique IDs for each drug and adverse reaction. This allows the data mining step to be run efficiently. This step outputs three files: a mapping between drug names and unique IDs, a mapping between adverse reaction names and unique IDs, and the copy of the table of adverse reaction reports using unique IDs instead of drug and reaction names.

The data mining element, implemented in C++, takes the preprocessed FAERS data and finds patterns in the data that may be associated with **MDARs**. To accomplish this task, the **FP-growth** algorithm is implemented to find the complete set of frequent patterns in the preprocessed data. After the frequent patterns have been generated, the program outputs two files to be used as input to the next element in the backend. The first file contains the frequent itemsets of drugs and their support values. Meanwhile, the second file contains rules indicating a potential causal relationship between a drug combination and an adverse drug reaction.

After all the rules are generated, they can be ranked using the Contrast Score. However, due to size of the dataset, it would be inefficient to calculate the score for every single rule reported in the last step. Therefore there is a filtering step that ignores every rule that is entirely contained by another rule.

For example, this filtering step would ignore a rule that suggests Aspirin and Excedrin lead to nausea if there is also a rule that suggests Aspirin and Excedrin lead to nausea and headaches. This prevents needless computation.

The final step in the MARAS program is generating results for use in DIVA. The final program, Explorer, sorts the list of rules by their contrast score, returning only the highest scoring. It creates three reports, one which contains the MDARs and all associated information in a human-readable format, one which has the support, confidence, and score in three comma-delimited columns, and a third which contains the same information as the fist file in a comma-delimited format.

2.2. DIVA Methodology

DIVA (Kakar and Qin 2017–2018) was developed in consultation with evaluators from the FDA to ensure a complete understanding of the current drug safety review process and the challenges associated with analyzing Drug-Drug Interactions. These evaluators are domain experts in the field of drug-safety research with focuses on multi-drug interactions. The consultations with these evaluators helped Kakar and Qin develop a list of requirements for an application that could successfully assist in safety research in multi-drug interactions. The requirements from Kakar (2016) are discussed below.

Listing 2.1 DIVA Requirement 1

```
Provide an overview of all Hypothetical Drug-Drug Interactions (HDDIs)
```

The simplest requirement is that evaluators need to have a method for easily scanning for interesting HDDIs. They need to be able to identify those interactions that *are* heavily supported by the data, so that they can focus on those. That led to the second requirement:

Listing 2.2 DIVA Requirement 2

```
Facilitate the detection of interesting HDDIs
```

This means that, in the visualization, hypothetical interactions that are well-supported in the data must be visually distinct from those that are not well-supported. This distinction must be clear enough that the most interesting HDDIs are easy to separate from the least interesting ones. This requirement also seems to encourage the ability to filter away the least interesting HDDIs.

Listing 2.3 DIVA Requirement 3

```
Enable analysts to change importance criteria for HDDIs interactively
```

Continuing the idea of filtering, this requirement shows that the different ways that evaluators could prioritize the hypothetical interactions must all be a part of the visualization, and switching focus between them must be easy.

Listing 2.4 DIVA Requirement 4

```
Facilitate detection of severe ADRs
```

Those reactions that are immediately life-threatening are more urgent than those that only cause mild discomfort, so it is important to find the most dangerous ADRs, so that they can be prioritized.

Listing 2.5 DIVA Requirement 5

```
Facilitate prioritization of drugs
```

The previous requirements also centered around the concept of **prioritization**, where evaluators wanted to not only filter away uninteresting hypothetical interactions, but also to have a way to choose an order for evaluators to examine interactions. This would allow for a project manager to choose an order of prioritization for a team of evaluators to work on.

Listing 2.6 DIVA Requirement 6

```
Link to underlying reports
```

Without access to the underlying report data, the visualization is hard to draw any conclusions from. There is significant information useful to evaluators in the reports sent to FAERs, and that needs to be accessible from the visualization.

Listing 2.7 DIVA Requirement 7

```
Support custom annotations on MDARs
```

If the evaluators want a single application to provide the entire toolchain for MDAR analysis, notetaking is indispensible. Requirement 7 allows for evaluation periods to take longer than one sitting, and evaluators could save their notes between sessions.

Listing 2.8 DIVA Requirement 8

```
Support smooth and interactive exploration of FAERs data
```

Usability is important for any new application that gets introduced to a work environment. Requirement 8 reinforces that priority.

Listing 2.9 DIVA Requirement 9

```
Use familiar visual metaphors and respect the user's mental model about
drugs and ADRs
```

A significant part of usability is concerned with the learning curve for new users. Part of minimizing the difficulty of learning a new system is relying on metaphors that the users would already be familiar with, either from applications they use in other areas of their work, or in daily computer use. That is the purpose of Requirement 9.

In order to fulfill these requirements, DIVA uses four main interactive views: Overview, Galaxy View, Profile View, and Reports view. As discussed in Sec. 4.4, we renamed each of these views to more closely reflect the user interactions associated with them.

2.2.1. Overview View

The Overview allows users to analyze MDARs at a macroscopic level; this view shows the entire space of Hypothetical Drug-Drug Interactions (HDDIs).

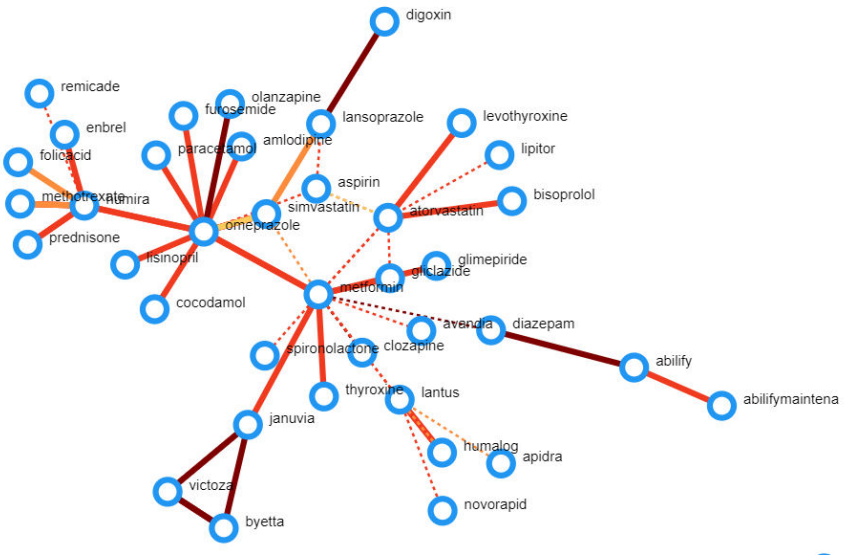


Figure 2.2.: Overview tree

This view utilizes a node-link diagram to visually display relationships between drugs and ADRs. In this view, nodes represent drugs, and edges represent an interaction between two drugs. If the edge is a dashed line, the interaction is known; if the edge is a solid line the interaction is unknown. The color of each edge represents the contrast score of the interaction.

2.2.2. Galaxy View

The Galaxy View allows analysts to quickly get an overview of MDARs associated with a specific drug.

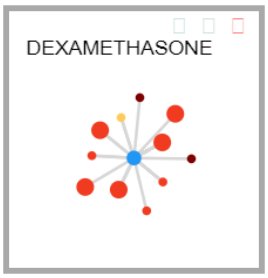


Figure 2.3.: Galaxy View

The center node of the Galaxy View represents the drug of interest, while the nodes surrounding it represent the drugs that interact with that drug of interest. The color of the box surrounding this view indicates the count of severe ADRs associated with

this drug. Users can view multiple different drugs at a time, sorted by name, interaction count, or severity as necessary. The size of each individual node indicates whether any of the MDARs between the two drugs are unknown; if there exists at least one unknown MDAR, the node will have a larger size than those with no unknown MDARs. Users can find additional information about each interaction by hovering over them.

2.2.3. Profile View

The Drug Profile View provides a more detailed look at an individual drug in the form of a modified tree layout, consisting of three levels. The root node represents the selected drug, the second level displays all of the drugs that interact with the selected drug, and finally, the third level represents the ADRs that exist between the drugs. Normal ADRs are represented with a tan color, while severe reactions are labelled purple.

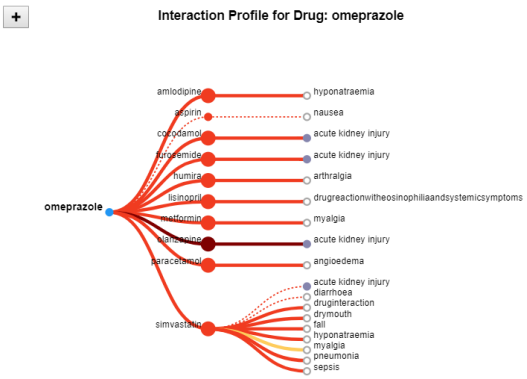


Figure 2.4.: Profile View

2.2.4. Reports View

The Reports View allows analysts to view the raw data directly. This view shows a list of all reports related to a chosen drug or drug interaction, enabling the user to see the underlying information related to a drug or ADR.

2.3. Minimum Viable Product

In Lean Startups, a minimum viable product (MVP) is "an experimental object that allows for empirical testing of value hypotheses" that has been "built with a minimum amount of effort and development time" (Münch et al. 2013, 138). A value hypothesis tests if a product is capable of being valuable to its customers once they are using it. Used

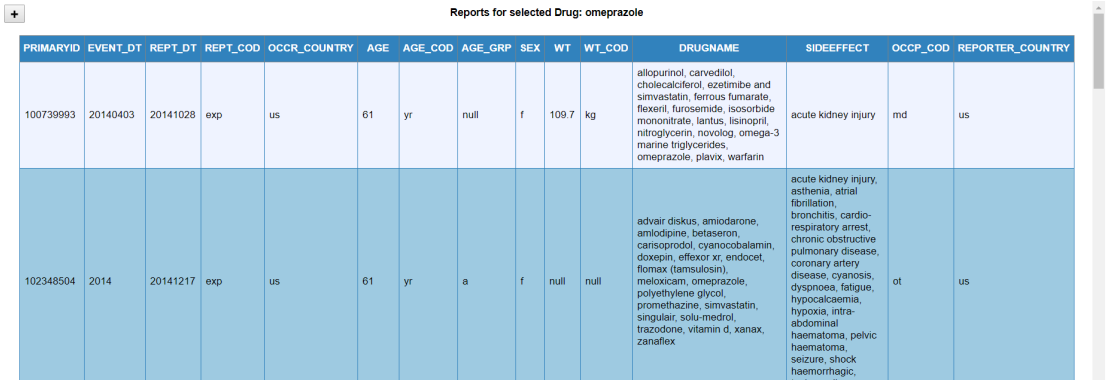


Figure 2.5.: Reports View

in conjunction, they create an experimentation tool that allows developers to determine if it is worth developing further.

Minimum viable products take many forms, depending on the type of project being tested. Some projects can test if they would get customer engagement with merely a landing page listing the planned features. In more established industries, it is often necessary to evaluate the value of a particular style of a feature and therefore a more developed MVP is required.

This project is focused on preparing the research conducted by Kakar and Qin for experimentation that can determine the value of their work to the FDA. We are creating a minimum viable product from the two research projects detailed above, to test the value hypothesis that an application that mines and ranks hypothetical drug-drug interactions from FAERS data and then serves a visualization fulfilling the requirements above provides valuable assistance in detecting and evaluating MDARs.

3. MARAS/DIVA System Review and Creating MIAP

The individual MARAS and DIVA tools relied on each other to be useful for analysis by FDA evaluators, our target use case for this project. DIVA requires data showing the drug interactions in order to create meaningful visualizations and MARAS, despite filtering and ranking the initial FAERS reports to focus on the interactions, creates far too much data to be able to effectively analyze without support from a visual interface. Thus it was vital that they be unified into a single system, which we called MIAP, the Multiple-drug Interaction Analytics Platform. The first phase of our project was to unify the tools, as they currently existed, into a single toolchain.

In order to accomplish this, we divided that task into two parts. Firstly, we reviewed the current codebase of both projects in order to decide on the cleanest way to integrate them. This involved a line-by-line reading of both systems. Then we refactored each project to be more maintainable, using published libraries, reducing the number of programming languages required, and focusing on applying standard software engineering practices to the development of each program. In essence, we rewrote each tool from scratch to ensure that they would work in concert without human intervention.

3.1. Analysis of Existing MARAS and DIVA Systems

In order to learn how the preexisting DIVA and MARAS systems functioned, the team decided to split the code up and perform in-depth analysis of the code and its functions. This decision was made because the preexisting code had little documentation and was difficult to understand without tracing the actions of the code line-by-line. Therefore, as part of the Deep Dive process, the team generated documentation to improve the code's readability and facilitate future efforts to refactor the code.

As the MARAS codebase was written in two languages (JAVA and C++), we divided the code review into two subtasks—one for analyzing the JAVA codebase, focusing on data transformation and score calculation, and the other concentrating on the C++ code for mining frequent itemsets and association rules. The two code-bases only interacted by reading the files created by the other applications and this lack of logical interactions between the two sections of the existing code made this division of labor sensible.

The codebase was further divided into 4 separately executed steps: Preprocessing, Rule

Mining, Closure Generation, and Score calculation. These correspond to the four steps specified in Sec. 2.1. The rule mining step was performed in C++, while the other three steps used JAVA code only.

The JAVA code was built around three main objects: `Itemset`, `Rule`, and `Group`. It also includes three executable classes: `Preprocess`, `ProcessFI`, and `Explorer`. As a whole, while there was no documentation, either inline, or at a JAVAdoc level, this module was fairly straightforward to analyze.

Of the three executable classes, `Preprocess` was the only one which did not interface with the data objects. After reading in the database of Adverse Drug Reactions, it created three files. The first two files were a unique mapping of the drugs and reactions referenced in the database. The third file applied that mapping to the database to create a list of reports where each report was made of the unique identifiers for each drug and reaction.

These unique identifiers are shared between drugs and reactions. For example, `Preprocessor` might map the drugs "Aspirin" to a unique ID of 1521 and "Excedrin" to a unique ID of 141, while mapping the reactions "Headache" and "Nausea" to 35413 and 33899, respectively. This would take a report line from the following format:

```
"Aspirin","Excedrin"    "Headache","Nausea"
```

The line would be output in the following format for consumption by the C++ data mining codebase, which operates using numerical itemsets.

```
1521 141 35413 33899
```

As shown above, this removes the indication that the drugs and reactions are different objects in the data. This was a particularly poorly documented detail of the process.

The C++ code in the existing MARAS system was used to generate frequent itemsets from the FAERS reports and output association rules that might indicate a causal relationship linking a set of drugs to one or more adverse reactions. The code uses the FP-Growth algorithm for frequent itemset generation.

Through completion of the C++ Deep Dive, many issues were discovered with the existing code. To begin with, there was barely any in-code documentation. This problem was fixed by adding Doxygen-style comments to describe each function and each class. There were also a number of redundant data structures defined in the C++ code that were also defined in the Java code. This problem raised the question of the feasibility of modifying the sections of code written in C++ or Java so that they are both in the same programming language and redundant data structures can be eliminated. Additionally, parameters such as the minimum support and the minimum confidence were both hard-coded constants. While this may work for testing or experimentation purposes, in reality users may want to specify these parameters to customize their results.

This figure shows the class diagram for the existing C++ code. Upon examination, it is evident that many of the classes represented in C++ are also represented in Java, leading

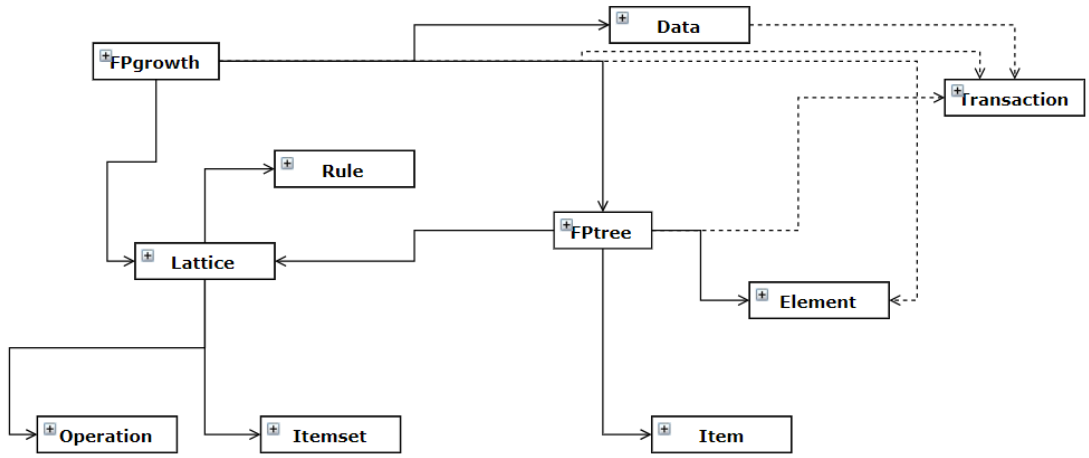


Figure 3.1.: C++ class diagram and object dependencies

to inevitable redundancies.

The results from the C++ code were used by the `ProcessFI` section of the JAVA code to populate a list of `Itemset` objects, where each different itemset length is organized together. It also populated a collection of `Rule` objects with the rules mine in the C++ code. These Itemsets were then filtered to create closures, using a method defined in `ProcessFI` and rules were further filtered to only be associated with Closed Itemsets. Additional filtering ensured that no rules originate with reactions.

The `ProcessFI` executable would output files containing each filtered rule, as well as all the closed itemsets. These files are then read in by the `Explorer` executable, which creates a `Group` objects. Each group object associates a closed rule with its component subrules (found from the unfiltered rule-sets). These are used to calculate the score by the process outlined in Sec. 2.1.

The D3 Visualization code produces the DIVA views described in Sec. 2.2. This code takes data produced by the Java rule-mining application and creates the resulting Overview, Galaxy View, Profile View, and Reports View. The Deep-Dive process for this code included the creation of comprehensive documentation using JSDoc, a Doxygen-style documentation system for Javascript code. By using JSDoc-style comments, each function could be annotated to describe its functionality and parameter values. This documentation can output HTML web pages that provide a quick overview of the organizational structure of the software, along with easy access to details on each function. In addition to documentation, the D3 Deep Dive involved changes to the coding style for consistency, including fixes to indentation, spacing and comments.

The SQL code in the MARAS system performs a variety of data manipulation and parsing functions which enable the MARAS system to utilize FAERS report data. First,

the SQL code contains functionality to parse FAERS report data from a `.txt` file format and construct corresponding SQL databases populated from that data. This code collects data on both drugs and reactions and combines them into a single table that contains only the information required by the MARAS system. There are separate functions to handle FAERS data from before 2015 and to handle the data from 2015 and beyond, because the format of the FAERS data changed slightly in 2015.

Second, the code includes functionality to connect rules mined by the MARAS system to the FAERS reports that the rules are generated from; for example, if MARAS discovered a rule stating that taking Advil and Warfarin together caused bleeding, this function would create a list of all of the reports that contained Advil and Warfarin in the drugs and bleeding in the reactions. This behavior is necessary to create the Reports View of the DIVA application.

When examining the SQL code, we considered several refactoring changes that could improve the usability and maintainability of the system as a whole. The codebase contains some functionality that could be accomplished equally effectively in Java, using APIs like JDBC. This would allow us to further centralize our code by keeping as much functionality in one language, and in one system, as possible.

To run the original MARAS system from start to finish, the user had to manually execute each step of the program. These steps included: pre-processing (Java), data mining (C++), finding closures (Java), and generating results (Java). At the beginning and end of each of these steps, the program would read from or write to disk in order to allow the program to pick up from where it left off in the previous step. This combination of excessive file I/O and an unnecessary reliance on user intervention made running the system unnecessarily complicated.

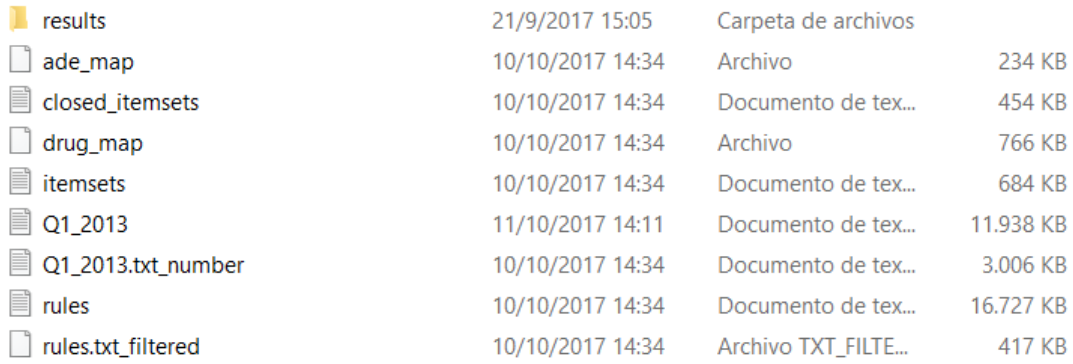


Figure 3.2.: Intermediate generated Files

This figure shows all of the files that are created to use during intermediate steps between the initial input and result generation. Following the program execution, these files are largely of no use to the user, other than possibly for debugging purposes.

3.2. JAVA Refactoring

Following the Java and C++ Deep Dives, the team discussed options for going forward. It was clear from the Deep Dives that the MARAS program should be modified to operate entirely in memory as a one-step process. This would make running the MARAS much simpler and make the process of integrating the MARAS system into the client-server architecture easier in the future.

Another major decision was whether or not to translate the C++ code into Java. In the end, the benefits of having all the code in one language outweighed the costs of rewriting that section of the MARAS system. This decision would make future maintenance of the system a much easier task, allow for the elimination of redundant data structures, and allow for a more natural one-step program flow that did not rely on file I/O to go between intermediate steps.

For the first iteration of refactoring the Java code, we decided to keep the underlying structure of the code as consistent as possible with the previous version while making modifications so that it ran entirely in memory. By minimizing the scale of the changes being made, we hoped to quickly produce a functioning prototype of the MARAS system that could be run on the server. Also in this revision of the code, we found and integrated Java implementations of the FP-Growth algorithm for frequent itemset generation and an association rule mining algorithm. The Java implementations of these algorithms were part of the SPMF Library put together by Philippe Fournier-Viger (Fournier-Viger et al. 2014).

After successfully refactoring three of the original four parts of the MARAS system to execute only in memory, it became apparent that the current program structure was insufficient. For one thing, the program did not take advantage of Java's object-oriented design principles, lacking much encapsulation. Instead, the existing code relied on frequent parsing of input files into objects, with most of the code operating on that object living outside of the object's class. Therefore, to improve the program's logical flow and make it more maintainable in the future, the team decided to refactor the MARAS system from the ground up, embracing the object-oriented design style to make the program flow more meaningful and readable.

Since we used new implementations of the algorithms for frequent itemset generation and association rule mining, we tested to ensure that both frequent itemset generation algorithms produced the same results given the same input and parameters and likewise for the association rule mining algorithms. To perform the test, both frequent itemset generation implementations were given the same transaction database as input and their minimum supports were both set to 10. Then, an output file was generated by both algorithms. These files were compared to test that they contained the same itemsets, regardless of itemset order or the order of items within the itemset. Similarly, the association rule mining implementations were both given the same set of frequent itemsets, a minimum support of 10, and a minimum confidence of 0. After receiving

output files from both algorithms, they were compared to ensure that each algorithm produced the same rules with the same supports and confidence levels, disregarding the location of rules within the file or items within the antecedent or consequent of a rule. As a result of this testing, the team confirmed that both algorithms produced equivalent results, indicating that the Java implementations had been successfully integrated into the program.

```
-----Itemset Test: SUCCESS---------------------------------------
File tested: ../data/itemsets_java.txt Correct File: ../data/itemsets_cpp.txt
Number of itemsets in correct file: 33782
Number of itemsets in test file: 33782
Correctly matched itemsets: 33782
Number of missing itemsets: 0
Missing itemsets:
Number of extra itemsets: 0
Extra itemsets:
Rule 374550 of 374550 Percent Complete: 100-----Rule Test: SUCCESS-----------------
File tested: ../data/rules_java.txt Correct File: ../data/rules_cpp.txt
Number of rules in correct file: 374550
Number of rules in test file: 374550
Correctly matched rules: 374550
Number of missing rules: 0
Missing rules:
Number of extra rules: 0
Extra rules:
2 test(s) completed. 2 test(s) were successful.
Test success rate: 1.0
```

Figure 3.3.: Results from rule Testing

For the second Java refactor, the team split the work into four distinct sections to reduce the time taken to perform the refactoring. The main idea of this refactor was to preserve encapsulation wherever possible, meaning that much of the existing code could be reused, but was moved within the class that it manipulated. Therefore, a set of rules could find its closure or filter out invalid rules, rather than performing these functions externally. The result of this refactor was a much simpler, more readable program flow. It was obvious which objects were involved in each step and what actions were being performed because the objects involved in the program's execution are well labelled and methods with descriptive names are called on the objects to perform any action.

3.3. Client-Server Setup

Our team's overall goal of the Client-Server setup process was to:

- Build a full-stack web application mainly for visualizing the data, providing a native-application-like user experience and supporting rich interactions with multiple components on the page.

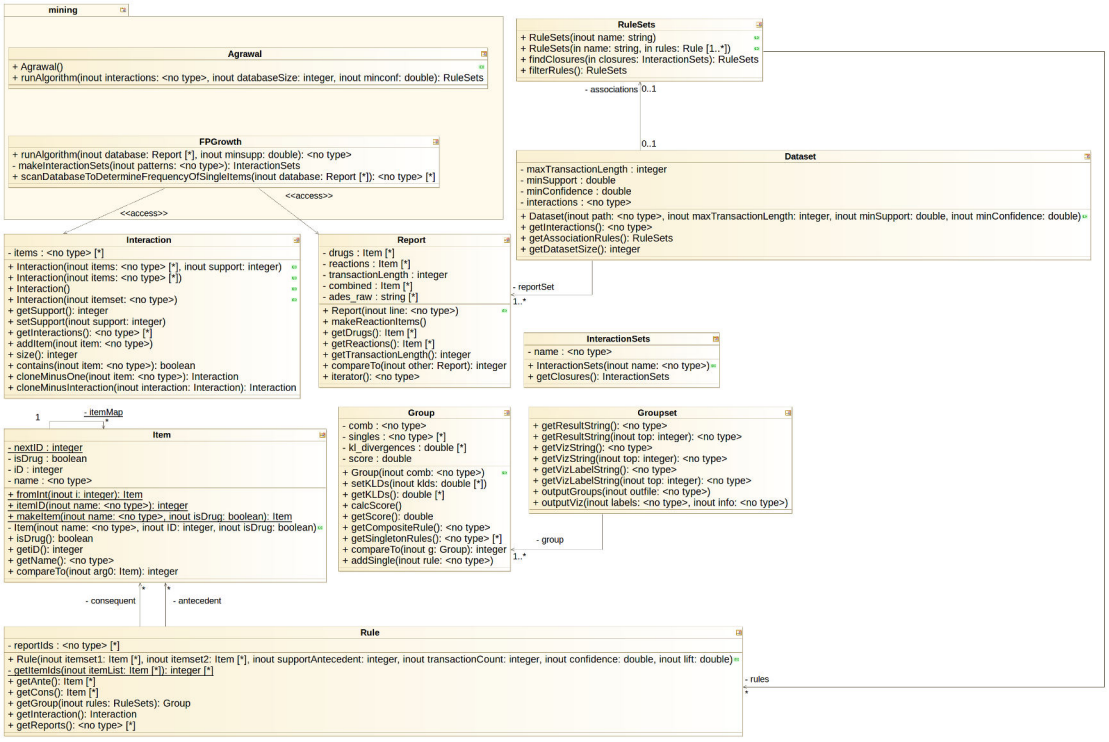


Figure 3.4.: Class Diagram for re-engineered JAVA Code

- Easy to setup with minimal configuration for local development and remote deployment.
- Modular so that different members in the team were able to work on different parts.
- Maintainable in order to continue to work productively as the application grew in functionality and complexity.
- Reliable in order to work correctly and avoid unpredictable behaviors.

3.3.1. Server-side Stack

We chose **NodeJS** (2018) as the runtime environment and **ExpressJS** ("Express - Node.js Web Application Framework" 2017) as our main back-end framework. There were two main reasons we used NodeJS. First, NodeJS uses Javascript as its main programming language. Therefore, by using NodeJS, we only had to program in one language across the front-end and back-end. Second, NodeJS is supported and backed by very active open-source community. Hundreds of thousands of packages and modules developed by the community are being hosted on **NPM** ("Npm," n.d.), the main package and dependency manager for NodeJS. ExpressJS is the most popular web micro-framework in the NodeJS community. It is fast, small and does not do much by itself. However, we could pull a lot of open-source modules in an Express application to make it useful and functional. Using NodeJS and ExpressJS as the foundation of the server fulfills the system's second goal. It is easy to build and start NodeJS server by running a few commands and setting environment variables.

We used **TypeScript** ("Get Typescript" 2017), an extension of Javascript as our main programming language of the server. It is completely backward compatible with JavaScript but provides nice object-oriented features such as classes and interfaces, and optional static type. The main benefit of using TypeScript is that it enables different Integrated Development Environments (IDEs) to provide extra features such as autocompletion, errors checking, refactoring. Using TypeScript allowed us to divide parts of the project into reusable and maintainable modules, which achieved the third and fourth goals as the project grew.

We used **Mocha** ("The Fun, Simple, Flexible Javascript Test Framework" 2018) and **Chai** ("Chai Assertion Library," n.d.) as the main testing framework and assertion library. They are the most popular testing stack in NodeJS community and proved to work well together. By writing tests and doing Test Driven Development (TDD), we were confident that every feature worked as we expected and not worried that refactoring or adding new features would break the code somewhere else. This fulfills the reliability goal of the system.

The server for this project served as a resource server for the client application. It provided different endpoints that allowed the client application to do basic REST operations such as reading, creating, updating and deleting on different resources. In order to do this, the ExpressJS server routing and middleware systems provided a set of API to handle

the client's HTTP requests. In this project, requests to read or retrieve the data were important because this allowed the client to get the data, analyze and visualize the information to the end users. For example, the client application could send a `GET` request to `/drugs/` to retrieve all records or `/drugs/{drugname}` to get all the records related to that specific drug.

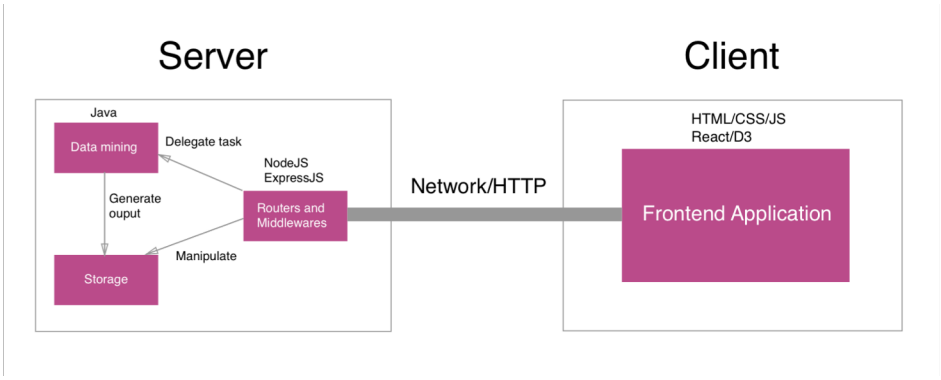


Figure 3.5.: Server Application Architecture

The Java data mining program was a submodule and ran in the NodeJS web server. Whenever the authorized user posts a new data file to the server, the server will process the input file with Java program and generate new output files for data visualization. The server also had a command-line interface to import the data files and run the Java program manually.

3.3.2. Client-side Stack

3.3.2.1. React

The Client-side stack fulfills the first and the most important goal of this project. We used *ReactJS* as the main front-end framework for building the interface of the application. It allowed us to build different components that we can reuse, which have their own HTML templates, behaviors and states. The components can be put together and interact with other components by sharing the state to make an interactive and native-app-like website.

3.3.2.2. Material UI

As mentioned above, one main advantage of using a modern Front-end framework like React is the ability to use third-party components and reuse components that we create. For the layouts and styling of the website, we exclusively used React components implementing Google's Material Design. By using these components, we can easily create a basic layout containing the navigation bar, sidebar, different panes, text fields, buttons and so on. These components have default style rules, but we are free to override them.

We can also attach event listeners to the components to make them interactive. Utilizing third-party components helps us focus more on the logic and behavior than on the styling and animation of the website while still keeping the application aesthetic.

3.3.2.3. Redux

Redux is a small state-management library that works well with React applications. It has a single store that stores all the current states of the applications. The idea behind Redux is simple: a React component (or any other kinds of component) emits a Redux action, which is a Javascript object that describes what happened. When receiving the action, the store singleton will decide how to change the state tree based on the reducers that you implement. Then, any components in the application that subscribe to the state change in the store will re-render themselves to reflect the change to the user. Fig. 3.6 briefly shows how React and Redux work together. Redux helps the application behave consistently and avoid bugs. It also makes the application scalable because when the application grows bigger and more states need to be controlled and shared among many components, Redux can manage all those states perfectly by creating more actions and reducers.

In the new DIVA application, Redux state management connects different components and their states. In Fig. 3.7, the Redux logging system shows different actions emitted by React components when the user clicks on a node in the Overview view. The store updates its state's property "selectDrug" to the selected node's drug name. It also requests the rules related to that drug from the server by the `REQUEST_DRUGS` action. When the `RECEIVE_DRUGS` action is emitted with the payload fetched from the server, the store updates its state's property `currentDrugs` by adding new drug rules related to that drug. The Galaxy View and Profile View, both of which subscribe to the state changes, will re-render themselves. The Galaxy View listens to the changes in the `currentDrugs` property in the state tree and renders a new graph corresponding to the recently added drug rules. When the Profile View sees changes to the `selectDrug` property, it will re-render and show the profile for the new drug. Redux also makes it easy to implement global filters and searching, simply by adding the filter states to the store. Those filters can be applied to all components that subscribe to the Redux store. Figs. 3.8, 3.9 shows actions related to updating the filter state.

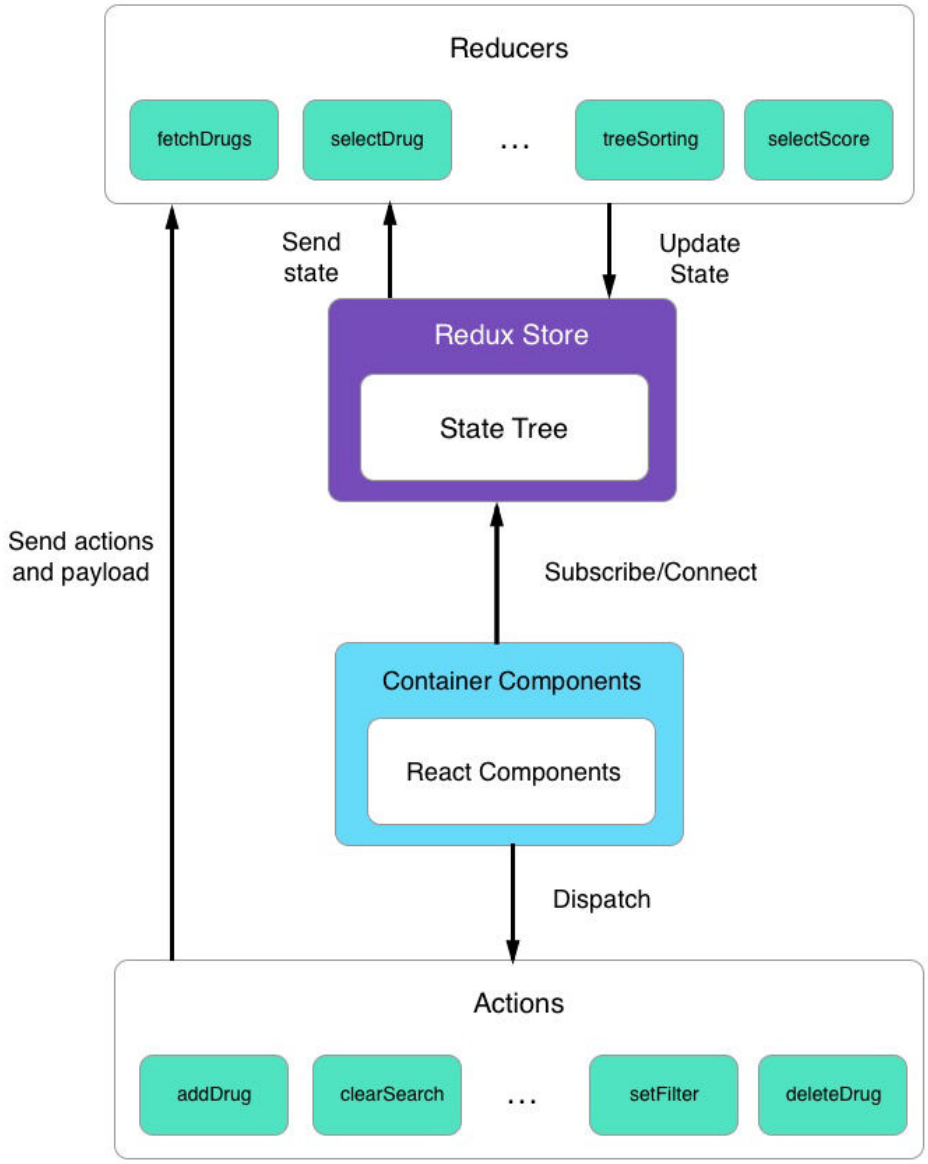


Figure 3.6.: Overview of React-Redux application architecture

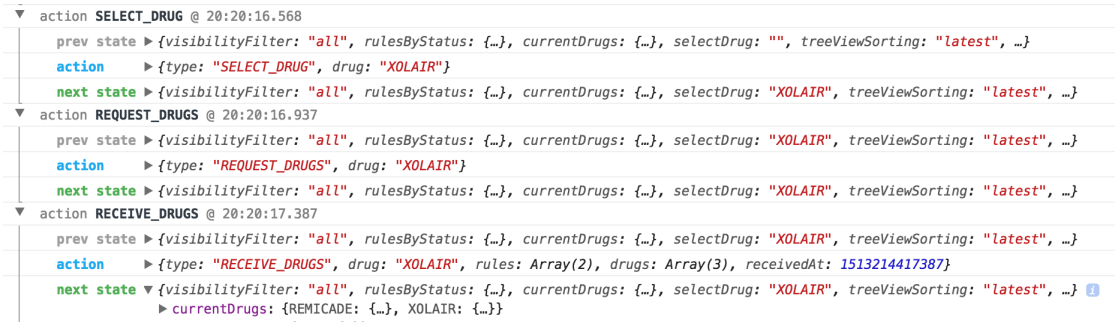


Figure 3.7.: Redux actions logging

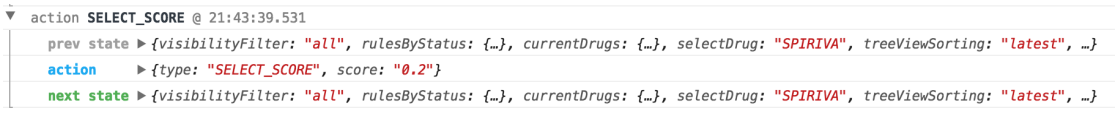


Figure 3.8.: Logging for filtering by scores

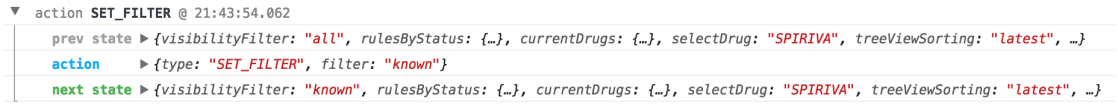


Figure 3.9.: Logging for setting a filter

4. Improving the MIAP Platform

After refactoring the existing MARAS code and DIVA visualization and creating our initial version of MIAP, it was time to begin adding in new features to meet the needs of a more complete integration of the two sides of the application.

We focused on four areas:

- Collective Data Mining for processing new input files when they arrive, in concert with the data already available for MARAS to analyze. This also includes integrating some of the preprocessing we did on the FAERS data to be able to read it into the MARAS program with less user intervention.
- Adverse Reaction Mapping to ensure that standard names and descriptors are used in reporting of the adverse reactions tied to FAERS reports, which ensures that known adverse reactions are matched whenever they occur.
- Drug Name Mapping to ensure that there is a standard vocabulary of drug names that prevents trivially different reports from being parsed as different drugs.
- Further improvements—refactoring and restyling—to the web interface to bring it up to a professional standard of usability and appearance.

4.1. SQL Refactoring and Collective Data Mining

Once the Java and C++ code of MARAS was successfully combined, we next worked on refactoring the SQL code that organizes and filters the raw FAERS data to get specifically the data that the MARAS system will use. Just as with the C++ code, we determined that it would be most effective to replicate the SQL code's functionality in Java; this would allow us to keep all of the MARAS code in one language, and all of the functionality would be contained in a single executable.

The process of parsing FAERS data evolved through multiple phases. First, the Java program was modified to take 3 file paths as input, corresponding to the three FAERS files MARAS uses: the drugs file (e.g. `DRUG17Q2.txt`), the reactions file (e.g. `REAC17Q2.txt`), and the demographics file (e.g. `DEMO17Q2.txt`). The program parses these three files to compile the data into Report objects, consisting of a list of drugs and a list of reactions; these reports are used by the Association Rule Mining algorithm to calculate the resulting rules.

The FAERS files are structured such that each line refers to a data point for a given

FAERS report. For the drugs file, each line contains the ID of the report, the name of the drug, and additional information not used by the MARAS system, such as the dosage. If a given FAERS report contained 5 drugs, the resulting drug file would contain 5 lines, each representing one of those drugs and referring to the same report ID. Similarly, each line of the reactions file contains the ID of the report and the reaction experienced, and each line of the demographics file contains the ID of the report and a set of demographic information about that report. The MARAS system specifically examines expedited reports, which are distinguished in the demographic file as "EXP".

To handle this data, the MARAS system must search through each of these files and match the lines according to their Report IDs. The result is a list of drugs, reactions, and demographic information that make up a complete report. From there, the system can filter out reports that are not expedited, and create Report objects from the drugs and reactions that can be used by the Association Rule Mining code.

After this change was implemented, we determined that it would be easier if the program directly uses the FAERS data as it is downloaded from the FDA's website: a .zip file containing a number of different files and folders, beyond just the three that the MARAS system uses. We refactored the input to now take in a single .zip file; it then extracts the files from this zipped folder and searches for the three relevant files, then runs the analysis on those files as before. This makes the command-line arguments for the file simpler, and reduces the amount of work that must be completed before running the MARAS code.

While this functionality was effective for mining rules from a single set of FAERS data files, which are organized by year and quarter (such as 2017 Q2), it did not allow us to find rules from the combination of multiple sets of FAERS data. Therefore, the next step in refactoring would be to support cumulatively analyzing sets of FAERS data in conjunction with the data already used. In order to accomplish this, the MARAS program saves the three FAERS files extracted from the zipped folder in a data directory, then parses all of the files in that directory into the Report objects. Therefore, when the user passes a zipped FAERS folder to the MARAS system, the data is added from that folder to the collection of data it has already been given, and then generates the ruleset with this new data included. This allows the system to generate rules based on any number of FAERS datasets collectively, as opposed to a single FAERS dataset at a time.

This setup successfully meets the goal of allowing the MARAS system to utilize multiple sets of data from FAERS, but it had a significant problem: repeatedly parsing the files is inefficient, and as more and more datasets are added the system took exponentially long to run. Indeed, even running the system on just two datasets, the 2013 Q1 and the 2017 Q2 data, took well over an hour to complete. Therefore it was important to find any way we could speed up this process; namely, by reducing the repeated work done each time the system ran.

The clearest candidate of repeated work that could be eliminated is the process of parsing

the raw FAERS text files. Each raw file contains a significant amount of data, much of which is ignored by the MARAS system. This resulted in a significant time cost that could be avoided by cutting out as much unnecessary data as possible. We determined the most effective method for this would be to store the parsed data in a new format after organizing the information into a set of Reports; when the system runs, rather than parsing the raw data over again, it will parse the new format that contains just the information it needs. The format we used is a text file where each line represents a report; each line contains a list of drug names and reaction names, with an arrow symbol "->" in between to distinguish them. Additionally, each drug or reaction name is separated by a comma. This results in concise data that only stores the information needed by the system. This dramatically reduced the size of the data being parsed; while the raw FAERS files for 2017 Q2 combined were almost 200 MB of data, the simplified file for that same data was just 16.5 MB. The results were also clear in terms of the time the system took; whereas running the system on the 2013 Q1 and 2017 Q2 data took over an hour before the change, after the change it took around 12 minutes from start to finish.

Next, there were several steps that made the MARAS system more efficient and simpler to use. First, the command line argument format was refactored to emphasize simplicity and flexibility. While the first version of the MARAS system simply defined command line arguments based on the order of the arguments (the first argument was the FAERS file, the second argument was the output path, etc.), the new version used specific flags to indicate which argument was which, such as "-f" to indicate the FAERS file(s) used and "-o" to indicate the output path. This meant the user would not have to memorize the order of the arguments, and also allowed for more flexibility for certain parameters. For example, under this format the MARAS system supports the input of multiple FAERS files at a time, which provides greater flexibility.

4.2. Adverse Reaction Mapping

One important component of the DIVA/MARAS system is the ability to label potential drug-drug interactions that are mined from FAERS data as *known* or *unknown*. A drug-drug interaction is labeled as *known* if the interaction has been previously documented. Otherwise the rule is labeled as *unknown*. This labeling process is important because it allows FDA analysts to filter out interactions in the DIVA visualization that correspond to known drug-drug interactions so that they can instead focus on discovering novel ADRs.

After researching sources of known drug-drug interactions, the team considered two potential sources: Twosides and a RESTful API that draws information from the ONCHigh and Drugbank databases (Tatonetti et al. (2012), ("Drug Interactions API" 2017)). Twosides is a database available as a file download that contains simple known drug-drug interaction rules, each rule consisting of two drugs and one adverse reaction. Meanwhile, the RESTful API takes as input the name of a drug and provides information

about any known interactions with other drugs. However, the problem with the reaction information obtained via the RESTful API is that it provides a general description of the ADR, rather than a descriptive term such as those found in the FAERS reports and the MARAS rules. For example, for the reaction between Vincristine and Cyclophosphamide the RESTful API describes the ADR as "The serum concentration of Vincristine can be increased when it is combined with Cyclophosphamide", whereas corresponding Twosides and MARAS rules label the ADR as Anemia. Therefore, the RESTful API could not easily be used to decide whether a given MARAS rule is known or unknown due to this difference in the method of describing ADRs. For this reason, the team chose to use the Twosides database for the purpose of identifying known drug- drug interactions.

After finding this source for known drug-drug interactions, there were still problems with matching the drug-drug interaction rules found in the MARAS system with the known rules from the Twosides database. The main issue was that the names of ADRs in the Twosides database did not always match up with the names of ADRs used by the FDA in the FAERS data. This meant that there were rules mined by the MARAS system that were equivalent to rules found in the Twosides database of known drug-drug interactions but were not matched because the term used to describe the MARAS rule's ADR was a synonym for the term used to describe the Twosides rule's ADR. Consider the following example. The MARAS system mines the rule that states that Aspirin, when used in conjunction with Metoprolol, causes Myocardial Infarction. Meanwhile, a similar rule in the Twosides database states that Aspirin, when used in conjunction with Metoprolol, causes Heart Attack. While these two rules are equivalent in meaning, the MARAS system failed to label the MARAS rule as known since the ADR names were different. This problem supported the need for ADR name standardization between the known rules in the Twosides database and the rules mined from the FDA's FAERS data.

The MedDRA hierarchy is a commonly used form of classifying and grouping related terms for ADRs (("MedDRA Hierarchy - How to Use" 2017)). The hierarchy is organized from most general to most specific. The levels of the hierarchy are System Organ Class (SOC), High Level Group Term (HLGT), High Level Term (HLT), Preferred Term (PT), and Lowest Level Term (LLT). LLTs are the most specific terms and describe how an observation is usually reported. Each LLT corresponds to only one PT, and each PT corresponds to at least one LLT. In addition, PTs can have synonyms. The next level of abstraction groups similar PTs under a single HLT, which are further abstracted into HLGTs. Lastly, HLGTs are grouped into SOCs by the cause of disease, area affected by the disease, or purpose.

The ADRs given in the FAERS reports are in MedDRA PTs ("Questions and Answers on FDA's Adverse Event Reporting System (FAERS)" 2017). However, the ADRs in the Twosides database are either in LLT or PT form. Therefore, there is a need to convert all ADRs to PTs and recognize when two or more PTs are synonyms. These are the principal goals of ADR name standardization in this context. However, it should also be noted that the proposed method of ADR standardization will also help to reduce the number of MARAS ADRs that are synonyms for one another, resulting in rules with higher support

values that more accurately reflect the importance of the drug-drug interaction.

When researching methods of ADR name standardization, the team first discovered the SIDER database, which contains mappings from Lowest Level Terms to Preferred Terms that are extracted from drug labels Kuhn et al. (2016). To leverage this database for name standardization, each mapping was read in, using the process seen in fig. 4.1, to attempt to create ADR schemas corresponding to each distinct ADR. Each ADR schema has at least one LLT, at least one PT, and a standardized name (made up of the concatenation of all PTs in alphabetical order). An example of such an ADR schema can be seen in fig. 4.2; the Lowest Level Term arthritis has two Preferred Terms, arthritis and rheumatoid arthritis, and the standardized name "arthritisrehumatoidarthritis". Then, when an ADR term was read in, either from the FAERS data or the known rules, the term was checked against each ADR schema to see if it matched any of the LLTs or PTs of that ADR. If a match was found, then, as in fig. 4.3, the ADR term was changed to the standardized name from that ADR schema to be used in the subsequent steps of the MARAS system.

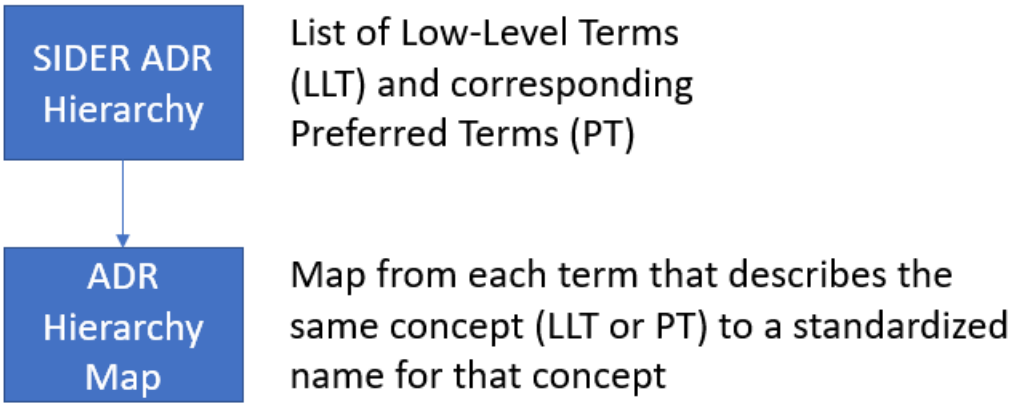


Figure 4.1.: SIDER map creation flowchart

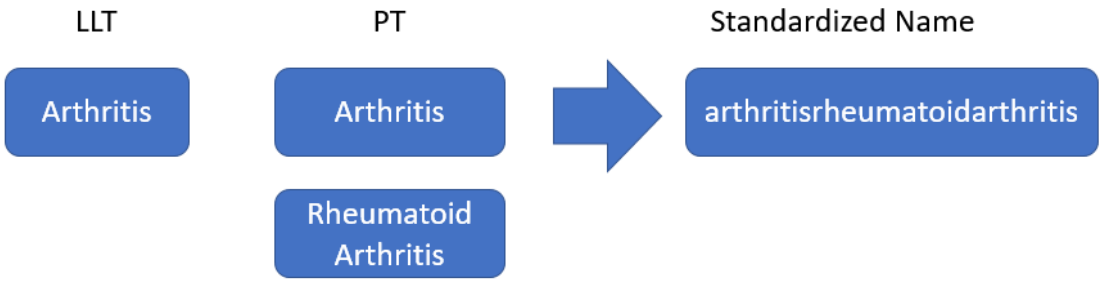


Figure 4.2.: Diagram showing the how standard names, LlTs and PTs are related

While the number of rules labeled as known by the MARAS system increased, the system still failed to recognize that Heart Attack is the same ADR as Myocardial Infarction

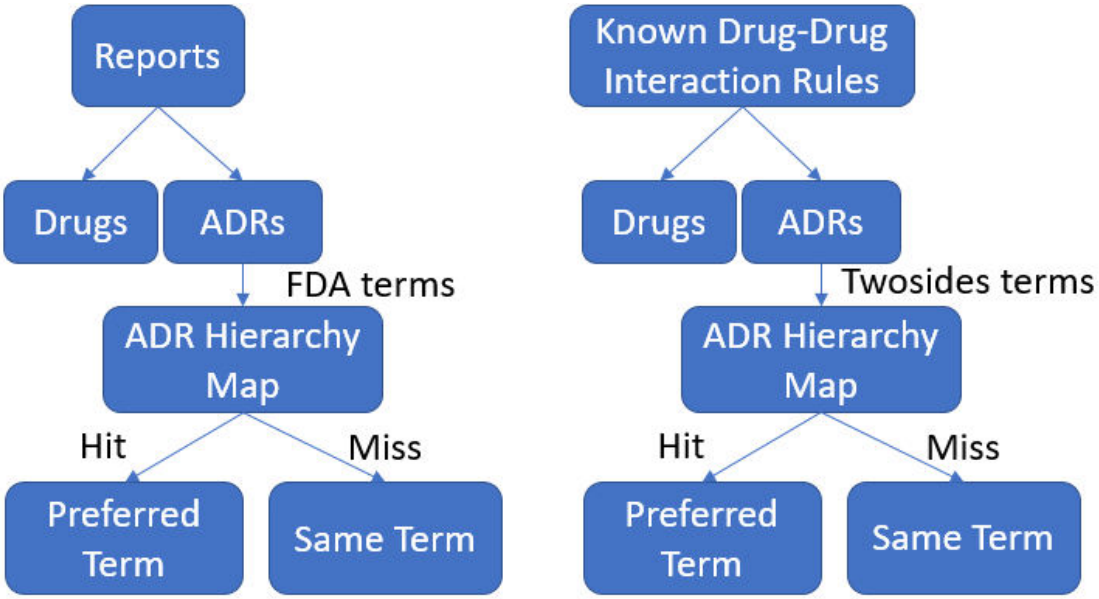


Figure 4.3.: Mapping Process flowchart for SIDER mapping

(Fig. 4.4), signifying that the SIDER database was incomplete in its mapping database from LLTs to PTs. Therefore, an alternative source of ADR name standardization, MetaMap, was explored.

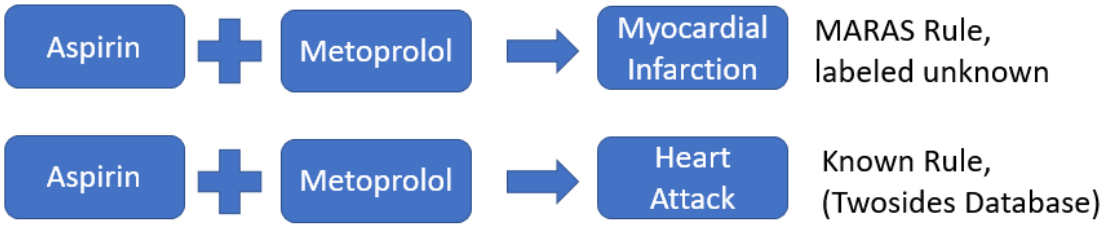


Figure 4.4.: Diagram showing an example where the SIDER mapping fails

MetaMap is a tool provided by the National Library of Medicine that parses natural language and identifies Unified Medical Language System concepts from the text (Aronson and Lang (2010), ("Unified Medical Language System (UMLS)" 2017)). To implement MetaMap into the MARAS system, every time an ADR term was read in from the FAERS data or the known rules, the term was checked against a "cached" map of all ADR name conversions that were previously made by MetaMap. This process can be seen in Fig. 4.5. In the event that the ADR term had not previously been converted with MetaMap, the term was submitted to MetaMap and the resulting PTs were concatenated to form a single standardized term for the ADR. This standardized term was then added to the map of "cached" terms to reduce future look-up times. An example of a mapping using MetaMap can be seen in Fig. 4.6.

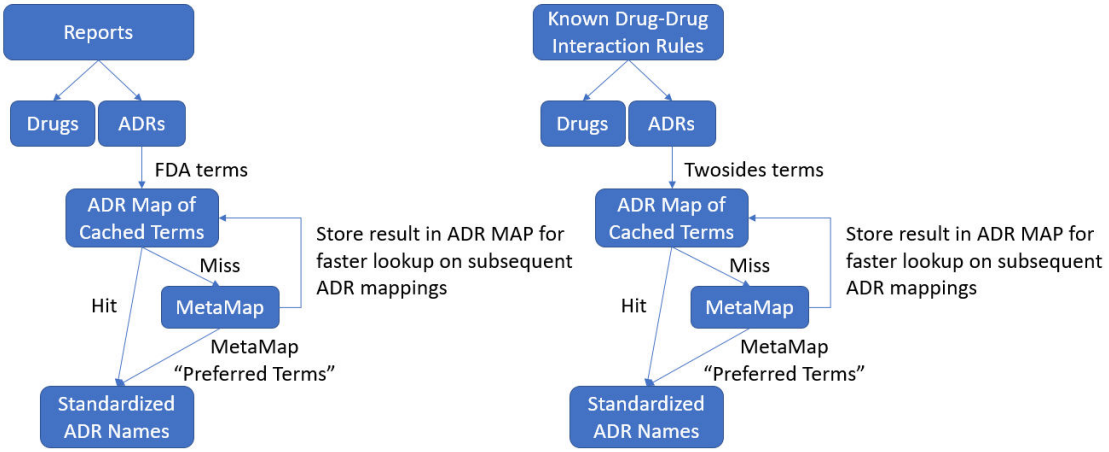


Figure 4.5.: Mapping process flowchart showing the MetaMap mapping process

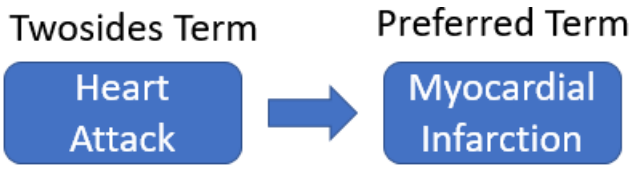


Figure 4.6.: Diagram showing an example mapping for MetaMap

After implementing both methods of ADR name standardization, a comparison, seen in Fig. 4.7, was done on the effect of each method on the MARAS process. Using the SIDER ADR hierarchy, the MARAS system detected 7657 distinct ADRs, whereas 7724 distinct ADRs were detected when using MetaMap. This may mean that the MetaMap system does not generalize as well as the SIDER ADR hierarchy. Despite appearing to exhibit less generalization, MetaMap outperformed SIDER at matching MARAS rules with equivalent known rules, finding 56 known rules out of 1542 while the SIDER-based system found only 13. In addition, the set of 13 rules labeled known by the SIDER-based system was a subset of the 56 rules found by the MetaMap-based system. It is also worth noting that the MetaMap-based system succeeded in recognizing that Heart Attack is a synonym for Myocardial Infarction, resulting in more rules being labeled as known, such as the rule in Fig. 4.7. As another part of the comparison, the time was measured during the process of standardizing the ADR terms for each of the known rules using both methods. A further experiment was conducted to measure the time taken to process the rules with and without parallelization for each method, as seen in tbl. 4.1. Without parallelization, the MetaMap-based method took 38 minutes to process the known rule database while the SIDER-based method took 45 minutes. With parallelization, the MetaMap-based method took 8.3 minutes to process the known rule database while the SIDER-based method took 8.6 minutes. Overall, the team decided that the MetaMap-based approach was a better match for the MARAS system because it not only provided for better

known rule matching but also outperformed the SIDER-based approach in terms of time efficiency.

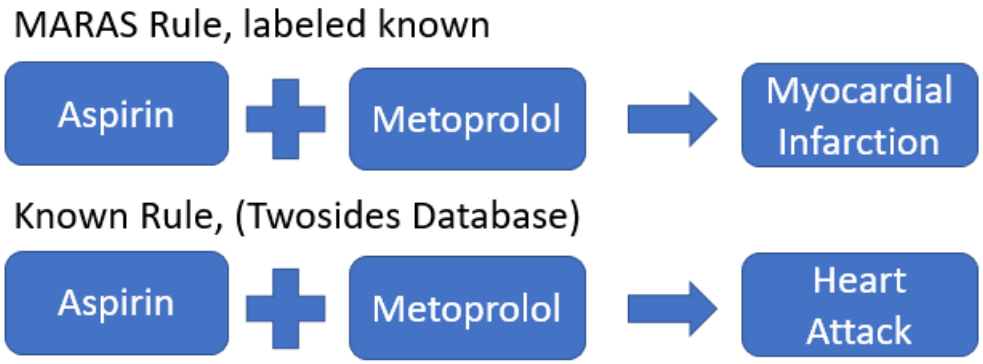


Figure 4.7.: Demonstrating that MetaMap resolves more matches than SIDER

Table 4.1.: Comparison of effectiveness and speed of ADR mapping methods.

	MetaMap	SIDER ADR Hierarchy
# of distinct ADRs	7724	7657
# of rules labeled as Known	56	13
Time to parse known rules	$\approx$ 8.3 minutes	$\approx$ 8.6 minutes
Time to parse before parallelization	$\approx$ 38 minutes	$\approx$ 45 minutes

Once MetaMap was chosen as the engine for adverse reaction mapping, we integrated not just the calls to the Metamap API, but also the operations of the MetaMap server. To successfully utilize the MetaMap API, MARAS uses three separate server instances: the SKR/Medpost Tagger, the Word Sense Disambiguation (WSD) Server, and finally the actual MetaMap server itself. Originally, this required manually starting these three separate server instances in different windows or as background processes, and then manually closing these instances when they are no longer needed. To improve simplicity and usability, this process was instead integrated as part of the MARAS system itself. The program starts the three separate server instances on its own, and closes them once the analysis is completed. This reduces the overhead required by the user to start the system, and makes the MARAS-DIVA integration simpler.

Finally, the MARAS system had to be deployed to our server, set up through WPI. Because the server is run through a Linux system, and our testing up to this point was performed on Windows systems, the first step required to integrate to the server was to handle Linux support. This primarily involved refactoring to properly support the Linux version of MetaMap; the MARAS system will detect which operating system is being used and launch the proper MetaMap instances appropriately.

4.3. Drug Name Matching

When parsing the input data in order begin preprocessing it for mining in our MARAS tool, we found that we had many similar drug names in our input fields, where a human would often be able to recognize as belonging to the same drug. However, in these situations the association-rule mining algorithms would end up treating them as different drugs, as their names wouldn't pass naïve equality checks. In order to accomplish this, our algorithms must consider a 'fuzzy' notion of equality.

This notion is often used in name lookups for people—as it is easy to misspell someone's name, especially in international settings. In their master's thesis, Du (2005), explored the effectiveness of various algorithms for matching similar personal names. They found that for language independent name matching, algorithms based on **edit distance**. gave the most effective results.

Edit Distance Given two strings a and b on an alphabet Σ (for example, the set of ASCII characters), the edit distance $\mathsf{d}(a, b)$ is the minimum-weight series of edit operations that transforms a into b.

Less formally, edit distance is the number of simple edits required to transform one string into another. The different operation classified as simple edits are the substitution of a letter, deletion of a letter, insertion of a letter, and the transposition of two adjacent letters (Du (2005) pp. 5).

The smaller the edit distance between two strings, the more similar each string is to the other.

When using edit distance algorithms to correct misspellings of surnames across multiple cultures, Du (2005) found that there was a tradeoff between speed and maximizing the number of relevant matches. They found that using a Bloom Filter and reverse edit distance would minimize the number of irrelevant matches while running faster, while using edit distance with a trie data structure and constant first letter would maximize the number of relevant matches at the expense of speed.

This sounds like it would work pretty well for our usage. While we are unlikely to encounter misspellings, the data we get from FAERS can often contain extra information, like the active ingredient in addition to the name, the dosage information, or even just a repeat of the drug names themselves.

However, in order to have anything to match, we need to have a database of normalized drug names—a set of drugs that we match everything back to. The main difficulty here is developing a vocabulary of drugs that is sufficiently large and will almost always contain the drug that was meant in the report. For this purpose, we are using RxNorm, a dataset of all perscribable drugs managed by the National Library of Medicine (Liu et al. (2005)).

In order for accurate sorting of the imperfect names in the ADR reports we need to have a dataset that includes almost all named drugs without having too many overlapping

names. Unfortunately, with RxNorm there were many overlapping names in even the smallest dataset provided, as the dataset provides some mapping functionality between different formats for each drug.

Thus, we had to further clean the dataset in order to make full use of it for drug name matching. For the first iteration of dataset cleaning, we used a script to create a JSON datafile that contained only drug names that didn't have another dataset entry with a name that was a substring of it's name.

For example, if there was `injectablediazepam` as well as `diazepam`, we'd only want to include `diazepam`. To accomplish this the following python loop was executed for every new drug name read into the script.

Listing 4.1 Naive Vocabulary Building

```
flag = True
for drug_id in output_dict.keys():
    drug = output_dict[drug_id]
    if name in drug:
        flag = False
        break;
    if drug in name:
        output_dict.pop(drug_id)
if flag:
    output_dict[nId] = name
```

This is just a naive check to see if our dictionary of drugs (and id's) either contains another drug whose name is fully contained within the current drug's name—which means we ignore this new drug—or if drugs we already added to the dictionary fully contain the current drug name, whereupon we remove those drugs from the dictionary and keep checking more drugs.

Of course, this makes parsing the dataset a very computationally expensive process, being $O(n^2)$ and having to process more than 200,000 lines. In initial testing, this takes almost an hour. Fortunately, once the dataset is satisfactorily cleaned, the drug list will not have to be recalculated for quite a while.

Unfortunately, this didn't do quite enough to remove redundancies in the dataset. This algorithm left overly-specific entries such as the following:

Listing 4.2 Vocabulary Sample Showing Redundancy from Dosage Information

```
1003676: 'zerit15mgoralcapsule',
1003680: 'zerit20mgoralcapsule',
1003684: 'zerit30mgoralcapsule',
1003688: 'zerit40mgoralcapsule'
```

A human user would reduce such drugs to `'zerit'`, leaving off the dosage information,

but since there wasn't necessarily an entry in the dataset merely saying `'zerit'`, the entries were never shortened.

Thus, we worked on creating a dictionary of strings that could be (mostly) safely ignored when parsing the drug names. This included words like `oral`, and `injectable`, but also numbers, units, or body part descriptors such as `vaginal`, `subcutaneous`. Because we aggressively removed matching character sequences from our potential drug names, we had to be sure that the patterns we were using wouldn't match unintended words. For example, without special treatment, the pattern `in` would match the final two characters in the word `protein`, which was undesirable. Thus, we made the pattern `' in '`, which would only match with whole words.

Additionally, drugs would often be of the form:

Listing 4.3 Vocabulary Sample Showing Redundancy from Alternate Forms

```
3858048: 'd4t30mgoralcapsule[zerit]',
3858049: 'dht30mgoralcapsule[zerit]',
3858259: 'd4t20mgoralcapsule[zerit]',
3858260: 'dht20mgoralcapsule[zerit]'
```

This is an easy special case, since we can just find the string inside the square brackets and reduce our result to `zerit`, which is our goal.

Once we developed this vocabulary, we can run the edit distance on all members of the vocabulary for a matchable drug name. Consider the following example vocabulary:

Listing 4.4 Sample Vocabulary for Edit Distance Example

```
vicodin
zamicet
ibuprofen
theophylline
```

If we are attempting to match `IBUPROFEN 500mg`, then the process would be to first perform the same normalization step as we did with the strings in the dataset: removing whitespace and converting to lowercase. Thus, `IBUPROFEN 500mg` becomes `ibuprofen500mg`. Then we calculate the edit distance for each of the strings in our vocabulary.

At first, we considered pure Levenschtein Distance, which is an edit distance involving only the substitution, insertion or deletion of single letters. We modified the cost of substitution to be the same as an insertion followed by a deletion, otherwise all words of the same length and no overlapping letters would be just as far away as a word that contains every letter in the source string, but is twice as long.

The evaluation methodoloy we used to test this distance was implemented in JAVA by Lloyd (1999). For example if we run `ibuprofen500mg`, we get edit distances of:

- `ibuprofen500mg -> vicodin`: 15
- `ibuprofen500mg -> zamicet`: 17
- `ibuprofen500mg -> ibuprofen`: 5
- `ibuprofen500mg -> theophylline`: 22

We also considered a second distance metric, namely the Damerau-Levenschtein distance, which adds the ability to swap adjacent characters. This operation needs to have a cost of at least half the sum of the costs of addition and deletion for the algorithm we found, implemented by Kevin Stern in 2014. We chose a cost of 1.2 as an initial test. For this metric, we get edit distances of:

- `ibuprofen500mg -> vicodin`: 15
- `ibuprofen500mg -> zamicet`: 17
- `ibuprofen500mg -> ibuprofen`: 5
- `ibuprofen500mg -> theophylline`: 19.4

In our example vocabulary, both algorithms come up with `ibuprofen` as the best match by far, which is good.

We ran this algorithm on the first 20,000 reports in our data set and tuned Damerau-Levenschtein distance parameters to attempt to maximize the number of distinct drugs mapped. We ended up with approximately 8500 distinct drugs matched in the first 20,000 reports using the following parameters:

- Add a character: 0.6
- Delete a character: 1.0
- Replace A character (add then delete): 2.0
- Swap two adjacent characters: 0.9

Once implemented and tuned, we needed to ensure that the behavior of our matching method was reasonable and did not consistently match unrelated drugs. To this end, we examined a random sample of the matched reports to document:

- The input name (e.g. `ibuprofen500mg`)
- The matched name (e.g. `ibuprofen`)
- The edit distance
- The number of times this mapping occurs

For example, a test sampling of 5 entries gave:

Table 4.2.: Table demonstrating evaluation sampling for Drug Name Matching

Mapping Number	Input Name	Matched Name	Edit Distance	Count
35498	`inexium`	`nexium`	1.0	27
515084	`carvedilol`	`carvedilol`	0.0	712
483937	`metoject`	`totect`	3.6	87
322073	`metamucil`	`metamucil`	0.0	45
397503	`cyclosporine un`	`cyclosporine`	4.0	2

Mapping Number	Input Name	Matched Name	Edit Distance	Count

The sensibility of a specific drug mapping is, however, a very domain-specific problem. It, for example, it doesn't make much medical sense to map `metoject` to `totect` as they are completely different drugs. Metoject is a cytotoxic antimetabolite used to treat severe arthritis, while Totect is a drug used to protect a patient's heart during chemotherapy for breast cancer.

Alternatively, INexium and Nexium are very close matches and are, in fact, the same drug. So this is a sensible match. For evaluation, we manually examined both sides of the mapping to determine if the match is sensible or not. However, not all sensible (or not sensible) mappings were created equal. It is more valuable that all 712 occurrences of Carvedilol in the reports were mapped correctly than that 45 occurrences of Metamucil were. Thus we weighted the value of each sensible or not sensible mapping based on how often they occurred.

Since we didn't want exceedingly common mappings like Carvedilol to completely eclipse the more niche mappings we decided to compare the weights of the square roots of the occurrences. This way the most common drugs are only somewhat more important than those that happen just enough to contribute to the rule lists.

In the above table, the four sensible matches (Inexium, Carvedilol, Metamucil, and Clyclosporine) contribute a weight of 40.0 to sensible matching, while Metoject contributes a score of 9.3 to the not sensible matching.

Therefore this sample gets a score of 0.811. Without the square-root weighting, this corresponds to 94% of drug instances being matched sensibly.

For a larger sample, consisting of 100 mapping instances, we got a score of 0.842, corresponding to 86% of drug instances being matched sensibly.

We then tuned the weights of the algorithm to try to maximize the effectiveness of our matching techniques. In the end, after 5 tuning passes, we were able to improve our sensiblility metric to 0.855, which corresponded to 91% of all drug instances being matched sensibly.

Our final tuning weights were:

- Add a character: 1.0
- Delete a character: 0.6
- Replace a character (add then delete): 2.0
- Swap two adjacent characters: 0.9

4.4. DIVA Interface Improvements

4.4.1. Improving Existing Features

We started by improving the design and functionalities of the DIVA website. Like DIVA, the new website has four basic views: Screening Overview (previously called Overview), Signal Triage View (Galaxy View), Signal Forensics View (Profile View) and Report View. These views are complicated and contain a lot of logic. Therefore, for better maintainability, we refactored them as a collection of React components which have parent-child relationships instead of one single component. The parent component can pass down the state to its child components as "props" and the child component can modify its parent's state by emitting some actions. Fig. 4.8 shows the layout of the website's main view.

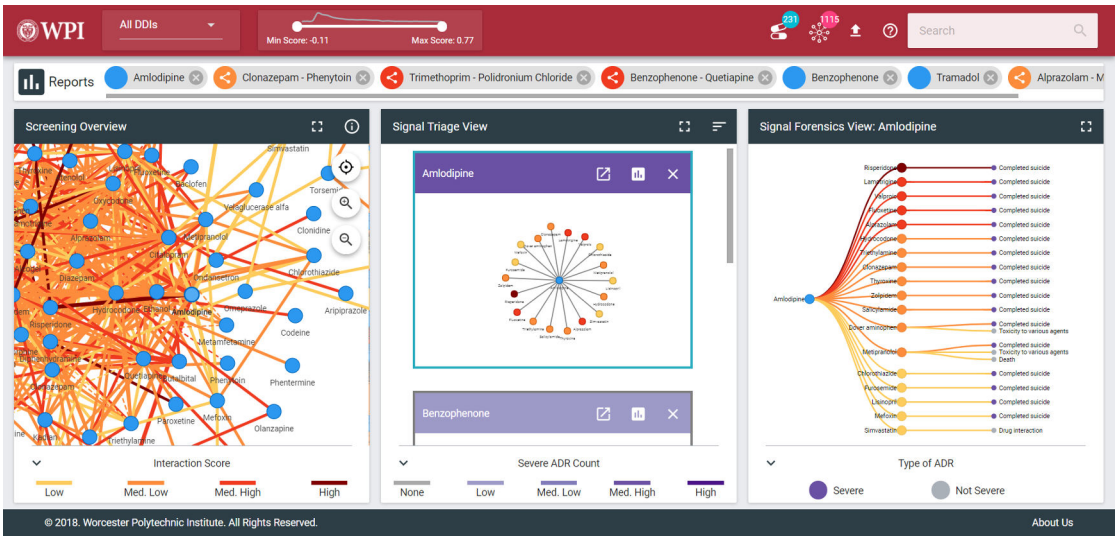


Figure 4.8.: Screenshot of DIVA main view interface

At the initial load of the website, the force-layout graph in the Screening Overview pane is displayed on the left. To render the force-layout network graph, we use the vis.js library. The library is highly customizable. We can customize anything from colors of nodes and edges to the duration of animation using a global "options" object which is passed to the React graph component. Drugs are displayed as nodes. Meanwhile, an edge represents the rule between two drugs with the highest score of all rules between those two drugs. There are two kinds of edges: the solid lines show rules that are *unknown* and the dashed lines show rules that are *known*. The colors of edges represent their contrast scores. We used four colors to show four uniformly distributed ranges of scores, labeled as: Low, Med. Low, Med. High, and High. In addition, by hovering over an edge, a pop-up appears and shows the information about the rule such as drug names, number of ADRs, highest scored ADR, score and status. By hovering over a node, the

corresponding drug name appears and all the edges containing that node are highlighted.

The Signal Triage View is similar to that of the old DIVA system. Each pane shows the interactions related to one specific drug, which is the blue node at the center of each graph. We implemented various sorting methods for the Signal Triage View: sorting by name of drugs, sorting by the latest drugs added to the view, sorting by the number of edges containing that drug, and by the number of severe ADRs. Each pane also has the ability to remove the drug from the Signal Triage View and show the drug's profile in the Signal Forensics View. One difference from the old system is that each pane's header is now colored to represent the severe ADR count associated with that pane's drug of interest. Previously, the border was colored to represent this information; however, in the new system a colored border is used to indicate the *selected drug*, or the drug that is currently being displayed in the Signal Forensics view and highlighted in the Screening Overview.

The Signal Forensics View and the Report View work as in the old DIVA website.

4.4.2. Implementing New Features

In addition to these views that existed in the old system, we have added new features to visually enhance the interface and assist users in leveraging the system to identify MDARs. We added a help section, improved the filtering functionality, created a chip system to help users track which drugs and reactions they wish to study the underlying reports for, provided summary statistics for the number of drugs and ADRs visible in the interface, and added a system for new FAERS data to be integrated into the analysis.

To begin with, a detailed help section was added. The help section features information about each of the different views, including the significance of different colors and line styles, as well as information about the different filters that can be used on each of the views. A screenshot of the help view is shown in Fig. 4.9.

The help section also includes a button that allows users to take a tour of the site. The interactive tour visits each view and each filter, offering the user the opportunity to try clicking on nodes and viewing the reports behind visualizations.

As seen in Fig. 4.10, the tour asks users to interact with the interface in order to progress on the tour, showing them the natural progession of tasks in the application while walking the user through the features.

Our filtering improvement focused on improving the usability of the contrast score filtering as well as filtering for specific drugs. Instead of filtering constrast score by using a minimum allowed score, we modified the filtering system to allow users to choose an upper and lower bound for the contrast scores of the ADRs they wished to see. Additionally, a curve above the range slider shows the distribution of contrast scores among all interactions. This information can help users to understand how they are

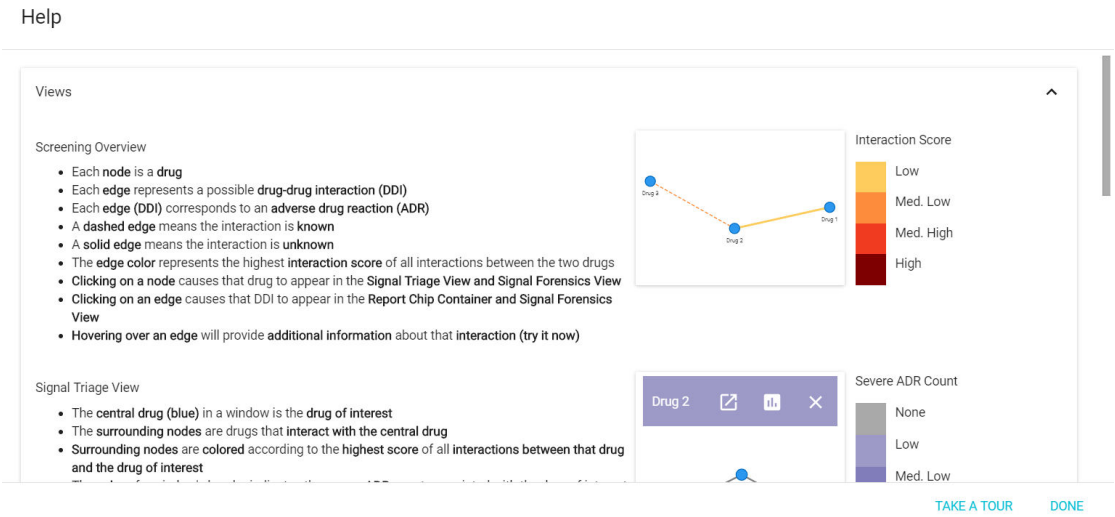


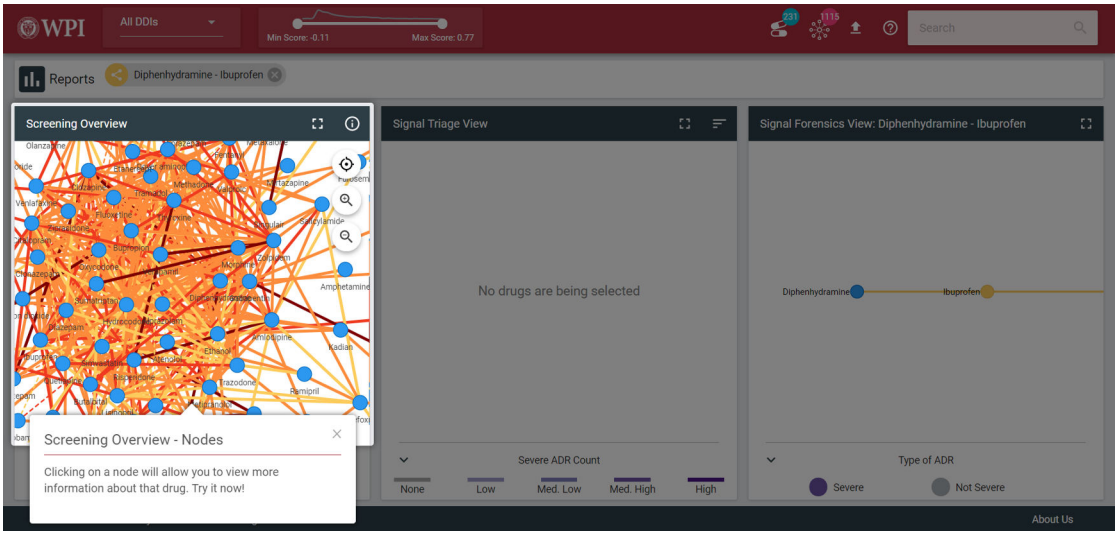
Figure 4.9.: Screenshot from Help Section

filtering the views and how much information they are filtering out. The new filtering widget is shown in Fig. 4.11.

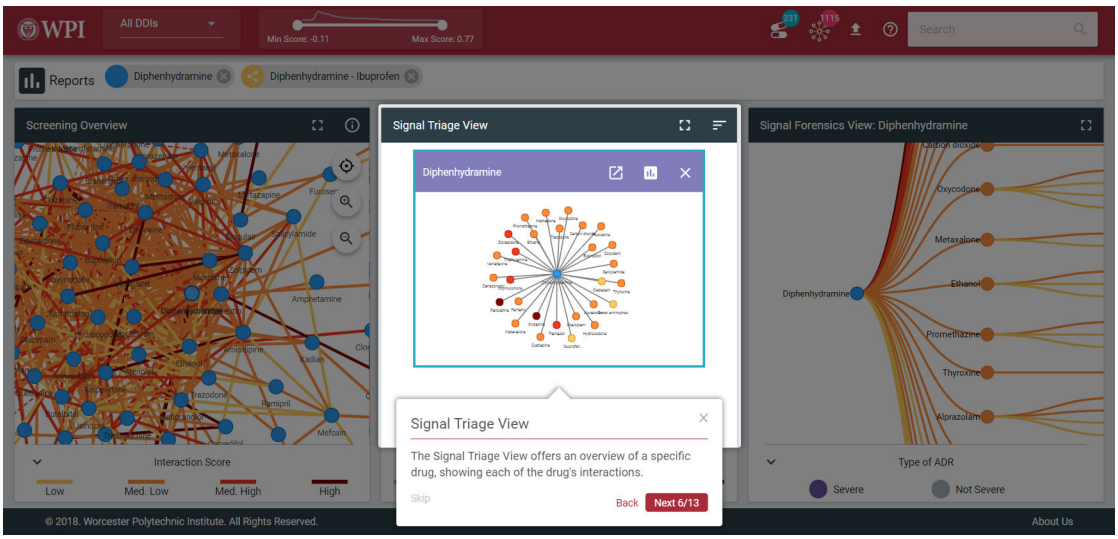
We also completely refactored the way that users could select specific drugs for further study. We replaced the list-view of drug names with a search box that supports the auto-completion of drug names so that the users need not remember or type the exact drug name as in the old DIVA system. Once the user clicks or presses enter, all three views will be re-rendered. The Screening Overview window zooms in to the searched drug. The searched drug is also added to the Signal Triage View and its profile is shown in the Signal Forensics View. A chip corresponding to the searched drug is also added (see below). In Fig. 4.12, we have an example of the auto-completion function on the search bar in action.

We added a *Chip* system to facilitate viewing reports. Whenever a user clicks on a node or an edge in the Screening Overview tab, a corresponding chip is added to the top of the screen. Chips are colored blue if they represent a single drug, or if the chip represents a DDI, then the chip is colored according to that DDI's score, just like the corresponding link in the Overview. Clicking on the chip will allow the user to view all FAERS reports that correspond to that drug or DDI. Note that to view reports for a drug, the user can also go through the Signal Triage View, but to view reports for a DDI, the user must use this chip system. In Fig. 4.13, we demonstrate the different colors that the chips can take.

We added drug and interaction icons to the toolbar at the top of the screen, seen in Fig. 4.14, to inform the user about the current number of drugs and interactions in the visualization. These numbers are automatically updated to reflect the current visualization when filters are applied.



(a) Tour Waiting for Action



(b) Tour After Interaction

Figure 4.10.: Screenshots of the tour

Figure 4.11.: Screenshot of Min/Max Score Filter

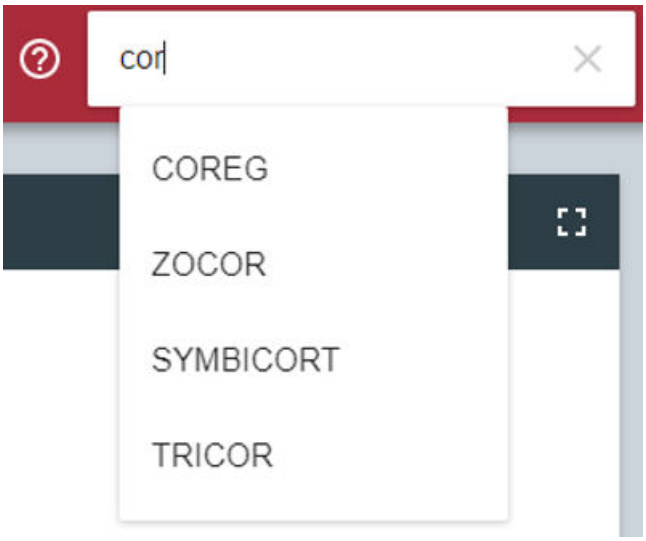


Figure 4.12.: Screenshot of Search Bar

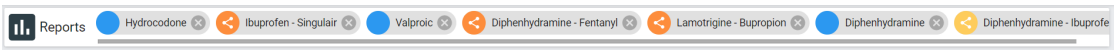


Figure 4.13.: Screenshot of Chip System

Figure 4.14.: Drug and Interaction Counts

Another key difference from the previous version is the addition of MARAS analysis status information and the ability to upload FAERS files and begin MARAS analysis through the user interface. As seen in Fig. 4.15, the user can follow the link to download zip files from the FDA's website. Then the user can upload one or more of those files to the system and start the MARAS analysis on those files. After the analysis is completed, the status information is updated with the status of the MARAS execution (successful, in progress, or exited with errors), a possible error message, a timestamp for the last time the information was updated, and a listing of all the datasets currently used for the visualization. Lastly, the MARAS system automatically updates the file used for the visualization, so users can see the results of the newly added data.

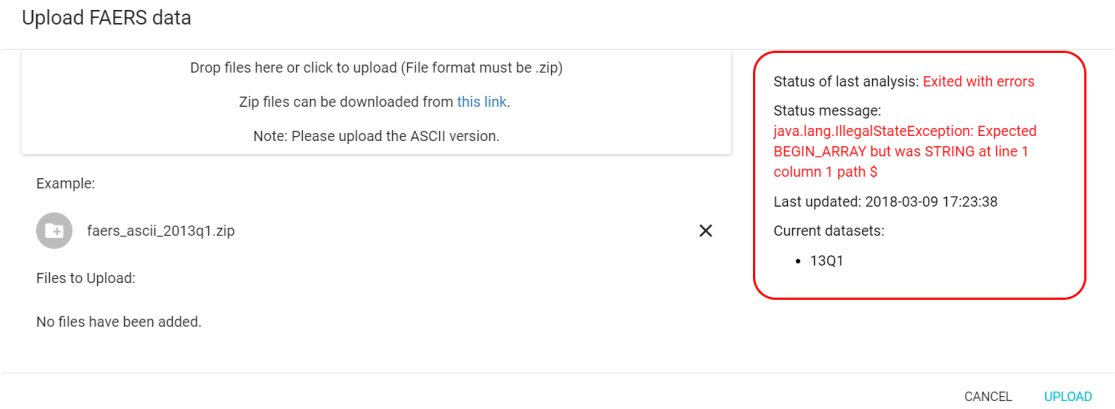
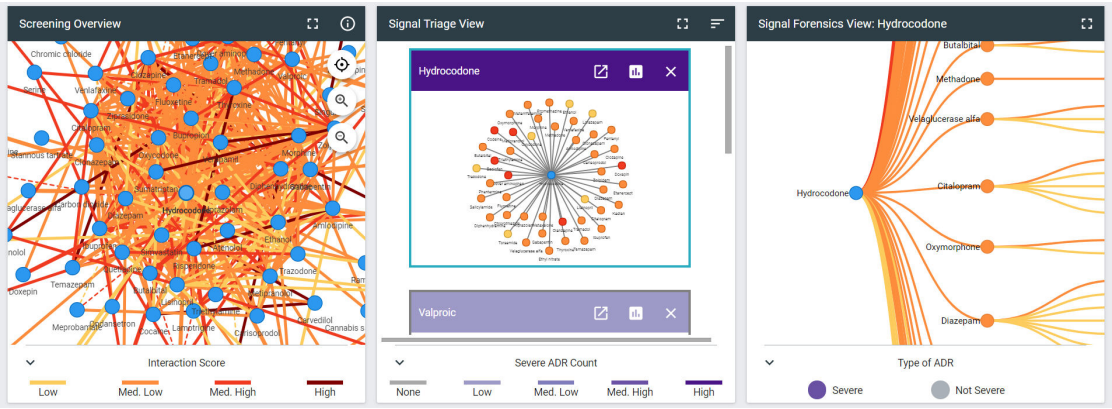


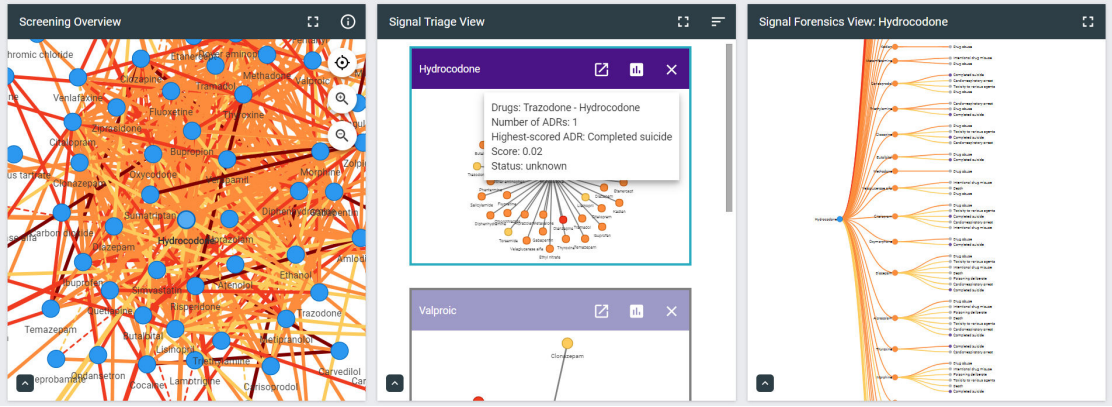
Figure 4.15.: Upload FAERS Files

We moved the legends showing the significance of the colors in the visualization beneath the view that they most correspond to. As seen in Fig. 4.16, there is also the option to hide the legends to provide more space for the visualization.

Lastly, we created an *About Us* (Fig. 4.17) section providing a summary of the team that developed DIVA, MARAS, and MIAP, as well as how to contact them.



(a) Legends Shown

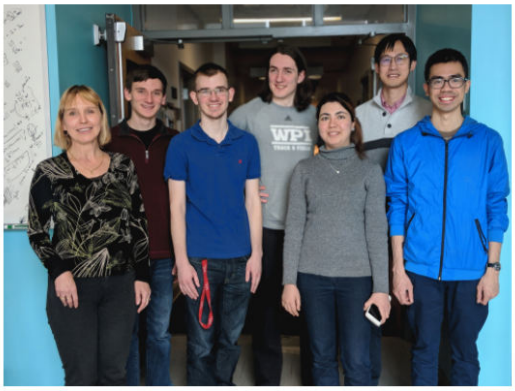


(b) Legends Hidden

Figure 4.16.: Screenshots showing how the legends can be hidden.

About Us

This system for analysis and visualization of multi-drug interactions was developed at Worcester Polytechnic Institute as part of a Major Qualifying Project. The project team was composed of:



Undergraduate Students:

- **Brian McCarthy**, Senior, CS '18
- **Andrew Schade**, Senior, CS '18
- **Huy Tran**, Senior, CS '18
- **Brian Zylich**, BS/MS Candidate, CS '19

Graduate Student Mentors:

- **Xiao Qin**, PhD Candidate
- **Tabassum Kakar**, PhD Candidate

Faculty: Prof. Rundensteiner and **Prof. Harrison**

FDA:

- **Suranjan De**, MS, MBA.
  Deputy Director, Regulatory Science Staff (RSS), Office of Surveillance & Epidemiology, CDER, FDA
- **Sanjay K. Sahoo**, MS, MBA
  Team Lead (Acting) Regulatory Science Staff (RSS), Office of Surveillance & Epidemiology, CDER, FDA

CLOSE

Figure 4.17.: About Us

5. Evaluation and Testing

After completing our integration and improvement steps in creating MIAP, we needed to ensure not only that it would run without bugs, but that it still was capable of providing users with the tools to accomplish all of the tasks that they were able to in the previous iterations of DIVA. This chapter, divided into our technical testing and user evaluation sections, will outline our evaluation processes.

5.1. Technical Testing

Though we started the project with existing algorithms for ranking and visualizing the drug interaction data, we redeveloped the systems from the ground up, so it was essential that we conduct testing to show that we were able to reproduce the functionality of the discreet tools we started with. To that end, we conducted several tests during our development process, ensuring that, at each stage, we were able to mine and rank the data correctly. Technical Requirement 1 formalizes our expectations for the technical performance of our system.

Listing 5.1 MIAP Technical Requirement 1

```
Correctly intake, mine, and rank new FAERS data provided by a user with no
further human intervention
```

The first part of this testing was ensuring that the mining and ranking systems, once implemented in early iterations of MIAP, matched the rule-generation results from the tools developed by our graduate student mentors. We developed a testing function that matched each section of the rule generation algorithm, itemsets generation, and rule generation, to ensure that, for the same data, our code generated the same output as the old system. In Sec. 3.2, we discuss the specifics of this methodology.

In Sec. **??**, we added several new features, including adverse reaction mapping, known-rule recognition, and drug name matching. When adding these features, we included tests to ensure that we correctly recognized known rules and that our drug name matching was sensible overall. The specific methodology of these testing procedures are discussed in detail in Sec. 4.2 and Sec. 4.3.

Since we were comparing two different algorithms for use as our known-rule recognition system, we were able to use the comparison metrics as our test for correctness. We

compared the number of rules recognized, trying to bring that number as high as possible without any obvious inaccuracies.

With edit distance for drug name matching, we first tried to find the vocabulary that had the most unique drug names without including a drug name that would be difficult to match because of dosage information, included alternate names, or had otherwise extraneous information in the drug name. The largest sensible vocabulary was the one we chose to move forward with. After finding the most suitable vocabulary, we moved on to testing the parameters for calculating the edit distance on each input drug name. Here the goal was to have the smallest number of insensible matches. We examined both the raw percentage of unique reports matched sensibly, as well as a metric that de-emphasized the most common matches, so that they would not overshadow inaccuracies with more obscure drugs.

5.2. Usability Evaluation

Following the development of MIAP, it was essential to confirm that the application is still capable of meeting the design requirements outlined in cooperation with the FDA. To this end, we decided to conduct a quantitative and qualitative user evaluation that assesses how well MIAP meets the requirements and identifies areas for improvement within the application. When constructing the user evaluation, we considered the traditional user study that involves comparing the new application with the state of the art. However, safety analysts at the FDA do not currently use any visualization tool for identifying MDARs, so we could not take this approach. While we considered conducting a comparison study between the previous DIVA interface and the new MIAP application, we decided to focus instead on having users complete a number of tasks within the MIAP application alone to focus more on its usability and less on the differences in visual appeal between the two interfaces.

5.2.1. Experimental Design Methodology

Our usability evaluation was focused on determining how well the MIAP application met the requirements from Kakar (2016), discussed in Sec. 2.2. During the development process for MIAP, we decided to skip over Requirement 7 and leave it for future work to focus on more pressing features.

Requirements 8 and 9 focused on usability and were general enough for us to consider part of the overall requirement for the project, in that creating an application that is usable is part of following good software engineering practices. Therefore, we omitted those requirements from our final list of requirements for MIAP. Thus, the six following requirements, adapted from Kakar (2016), informed the development of our usability study.

Listing 5.2 MIAP Usability Requirement 1

```
Provide a view of all ranked hypothetical interactions at-a-glance.
```

As in the original DIVA platform, viewing all the hypothetical interactions at-a-glance on the main screen is an essential requirement. Having designed the "Overview" view to be the root of all user interactions with the interface—it is where drugs and interactions are selected, filtered and searched for—we need to test how well the Overview provides this feature.

Listing 5.3 MIAP Usability Requirement 2

```
Facilitate differentiation between unknown hypothetical interactions
and already researched known interactions.
```

The mined data is going to include many reports for known drug interactions, so it is important to be able to tell them apart from the hypothetical interactions that need further research to confirm. We need to test if the method we used to differentiate them—solid lines for unknown, hypothetical, interactions, and dashed lines to represent known reactions—is sufficient for users to easily tell the two states apart.

Listing 5.4 MIAP Usability Requirement 3

```
Allow focus on select interactions for further study.
```

The Galaxy View and Interaction Profile are parts of the visualization that are designed to provide focus on a select set of interactions beyond what is available in the Overview. We need to be able to see if they can provide the needed functionality, in that they get rid of uninteresting or unimportant drugs and interactions and allow for users to be able to keep track of them.

Listing 5.5 MIAP Usability Requirement 4

```
Facilitate the prioritization of focused interactions by various criteria.
```

Prioritization is the key to being able to analyze a set of drug interactions. Once a user chooses or is assigned a group of drugs to examine, they need to be able to tell which is the most interesting, and that can depend on a number of criteria: the contrast scores of the interactions, the severity of the reactions, the number of reports for each interaction, or others. Therefore, we need to be able to test how well MIAP is able to prioritize interactions by different criteria.

Listing 5.6 MIAP Usability Requirement 5

```
Facilitate detection of severe ADRs.
```

As discussed in Sec. 2.2, we need to be able to see when an interaction involves a serious reaction. Adverse reactions that threaten imminent death are going to be prioritized higher than those which are less dangerous, so we want to be able to test how easy it is to notice those severe reactions.

Listing 5.7 MIAP Usability Requirement 6

```
Provide access to the underlying FAERS reports.
```

The underlying FAERS reports are the backbone of the evaluation process for confirming MDARs, and we need to test how easily available we have made those reports, as well as how accessible they are for the user to consume.

From these requirements, we developed a list of tasks that, if easily completed, would demonstrate that the MIAP platform had met those requirements. However, in its first iteration, this list included tasks that were easy for us, the developers, to complete in multiple views. We wanted our test to evaluate each view separately and show that the DIVA views added value to the system, so we refined our tasks to focus on one each view.

5.2.2. Experiment Overview

There were two planned phases for our qualitative evaluation experiment. First, we selected WPI students to interact with the interface in order to gain general insights of the usability of the system. Additionally, this would allow us to determine whether MIAP meets the requirements laid out in Sec. 5.2, which are based on those from Kakar (2016). Second, we intended to meet with domain specialists from the FDA to evaluate how well the tool meets their needs beyond the requirements they communicated with Kakar. However, we did not have time to meet with specialists from the FDA, so this part of the evaluation has been added to our recommendations for future work Sec. 7.2.4.

WPI Students To recruit WPI students for the study we sought volunteers vial email, social media, and word of mouth, offering participants a chance to win one of a small number of incentives.

After we had recruited participants, a member of our team met in person with each participant separately to conduct the evaluation. To begin with, the participant was given a brief outline of the methodology used to conduct the evaluation in order to ensure that they were still okay with participating. If they agreed to continue participating, then we asked for their email address to use in the case that they won an incentive. They were then given a short overview (Sec. A) of the purpose of the interface that we had developed along with some background knowledge necessary for a rudimentary understanding of how to interact with the interface.

Afterwards, the participant was presented with a series of tasks to complete using a set interactive version of the interface. While the participant was performing each task, they were asked to verbally indicate what they were trying to do and whether they were confused by anything. At the same time, the member of our team was taking notes regarding the participant's level of frustration/satisfaction, time taken to complete the task, and any verbal comments that the participant made.

Following the completion of the tasks, the participant was asked a series of questions

regarding their interaction with the interface and anything that they thought could be improved. The member of our team made notes as necessary. Then, the participant had the opportunity to ask any questions they might have had regarding our project. They were then notified that they would receive an email if they were later found to have won one of the incentives.

After all the WPI students had participated in our evaluation, a drawing was held to identify the winners of the incentives. Each student was assigned a unique number from one to the number of WPI students who participated. For each incentive, a random number generator was used to randomly select a WPI student to win that incentive. If the same student was selected multiple times, we repeated the aforementioned process until all incentives were assigned to different students. Then, each winner of an incentive was notified via email, and a meeting was be arranged for them to pick up their incentive.

FDA Domain Specialists Please note, we did not have time to meet with specialists from the FDA, so this part of the evaluation has been added to our recommendations for future work Sec. 7.2.4. However, in cooperation with PhD students Kakar and Qin we designed the following approach. To recruit domain specialists from the FDA, we would contact members of the CDER that have assisted the research of Kakar and Qin.

After finding domain specialists willing to participate in our study, a member of our team will meet in person with each specialist separately to conduct the evaluation. To begin with, the specialist will be given a brief outline of the methodology used to conduct the evaluation in order to ensure that they are still okay with participating. Then, they will be given a short overview of the interface that we have developed. Before seeing the interface itself, we will ask the specialist to try and imagine how they might use this kind of tool.

Afterwards, the specialist will be invited to explore the interface directly. We will guide them through the tasks that we set out for the WPI students, and ask them to try and use the interface to accomplish the use cases they had mentioned previously, as well as any other way they might intend to use it. While the specialist is working they will be asked to verbally indicate what they are trying to do and whether they are confused by anything. At the same time, the member of our team will take notes regarding the specialist's level of frustration/satisfaction and any verbal comments that the specialist makes.

Following this, the specialist will be asked a series of questions regarding their interaction with the interface, including whether they think the interface would help them in identifying adverse drug-drug interactions and anything that they think could be improved to help them better do their job. The member of our team will make notes as necessary. Then, the specialist will have the opportunity to ask any questions they might have regarding our project.

Both phases center around the requirements that we adapted from Kakar and Qin

(2017–2018), in the design process. The tasks we developed from those requirements each focus on their own views in the application. We presented the tasks with relationships to the views we wished the subject to focus on when accomplishing the tasks, so that we could understand how well each specific view fulfilled the requirements set out for them. Our tasks, organized by view, were as follows:

Overview (Req. 5.2)

- Task 1: Find 3 unknown drug-drug interactions (Req. 5.4, 5.5).
- Task 2: Find 3 drug-drug interactions with a score greater than **0.5** (Req. 5.4).

Galaxy View

- Task 3: Find and select drugs **Trazodone**, **Metaxalone**, and **Tylenol**. Of these drugs, which drug has the highest number of ADRs (Req. 5.3, 5.6)?
- Task 4: For the drug **Metaxolone**, how many high scored, (score $>$ **0.2**), and unknown interactions are there (Req. 5.4, 5.5, 5.6)?

Interaction Profile

- Task 5: Find which ADRs that occur between **Diazepam** and **Trazodone** (Req. 5.3).
- Task 6: How many reports support the interaction between **Diazepam** and **Trazodone** (Req. 5.7)?
- Task 7: Find all severe ADRs associated with **Trazodone**'s interactions (Req. 5.3, 5.6).

Report View (Req. 5.7)

- Task 8: Please open the reports for the interaction between drugs **Zoloft** and **Tylenol**.
- Task 9: List the three most frequent drugs reported with the interacting drugs **Zoloft** and **Tylenol**.

6. Evaluation and Testing Results

In this section we present the results of the technical and usability evaluations outlined in Sec. 5. Overall, the technical results confirm that the system correctly mines drug-drug interactions and show that the choices made for adverse reaction and drug name standardization are logical. Meanwhile, the usability evaluation results indicate that MIAP aligns well with the FDA's requirements while highlighting areas for improvement, some of which were promptly addressed.

6.1. Technical Results

6.1.1. Validation of MIAP Output

As seen in Sec. 3.2, after reimplementing the data mining part of the MARAS process in Java, we tested to ensure that the rules and itemsets generated by both systems were the same. Following testing with the same input data and parameters, we confirmed that the same 33782 itemsets and the same 374550 rules were generated, while accounting for an increased level of floating point precision.

6.1.2. Comparison of Techniques for ADR Standardization

Meanwhile, in Sec. 4.2 we compared the use of two methods for adverse reaction mapping and standardization. The first method used a dictionary of term mappings, where any term matching one of the keys in the dictionary would be changed to the value associated with that key. This approach used the SIDER ADR hierarchy. On the other hand, we tried using MetaMap, a tool provided by the NIH for recognizing medical concepts and returning standardized names. As a result of this comparison, we decided to utilize MetaMap to standardize adverse reaction names between FAERS data and our database of known rules. This is because, while MetaMap appeared not to generalize as well as the SIDER ADR hierarchy since it resulted in a higher number of distinct ADRs overall, MetaMap did outperform the SIDER ADR hierarchy significantly in matching known rules, finding 56 compared with the 13 found by the SIDER ADR hierarchy.

6.1.3. Quality of Drug Name Matching

As seen in Sec. 4.3, we conducted a series of 5 parameter-tuning passes to increase the performance of our drug name matching. Drug name matching was assessed using a random sample of 100 drug name mappings from each of the tuning passes. Once the random samples associated with each set of parameters was retrieved, we manually determined whether each mapping was "sensible" or not based on domain research and human observation. Then, the sensibility metric, described in greater detail in Sec. 4.3, was calculated for each sample, as well as the raw percent of mappings that were sensible. Through this process, we achieved a maximum sensibility metric of 0.855, which corresponded to 91% of the mappings being "sensible".

6.1.4. Time Breakdown Analysis

Finally, we conducted a time breakdown analysis on the complete rule mining process to pinpoint areas in need of improvement. Then, after optimizations such as caching, we again conducted a time breakdown analysis for comparison, as seen in Fig. 6.1. The time savings evident from this comparison are significant; whereas before the process took over 4.5 hours to run, the optimized process ran in just over 40 minutes. With this reduction in running time, we were able to more quickly debug future additions to the process, as well as drastically reduce the time that a user must wait to visualize new data once it is uploaded to MIAP.

6.2. Usability Evaluation Results

While there were originally two phases planned for the user evaluation, we chose to only conduct the user evaluation with WPI students due to time constraints. For this reason, this section focuses exclusively on the results from the evaluation with WPI students. The following results are based on user evaluations with 18 WPI students. These students varied in age from 18 to 22 years of age and all study Engineering. Of the 18 participants, 7 were female and 11 were male. These participants were acquaintances or volunteers.

The results from the user evaluation can be divided into user ratings and comments about specific tasks, user ratings and comments about more general elements of the user interface, and user ratings and comments about the visual appeal of the application.

6.2.1. Specific Tasks

After conducting the user evaluation, the results indicated that overall the application performed well, meeting each of its usability requirements except for Requirement 5.7. After each task, users rated the ease of use of the task on a scale of 1 (being very hard) to

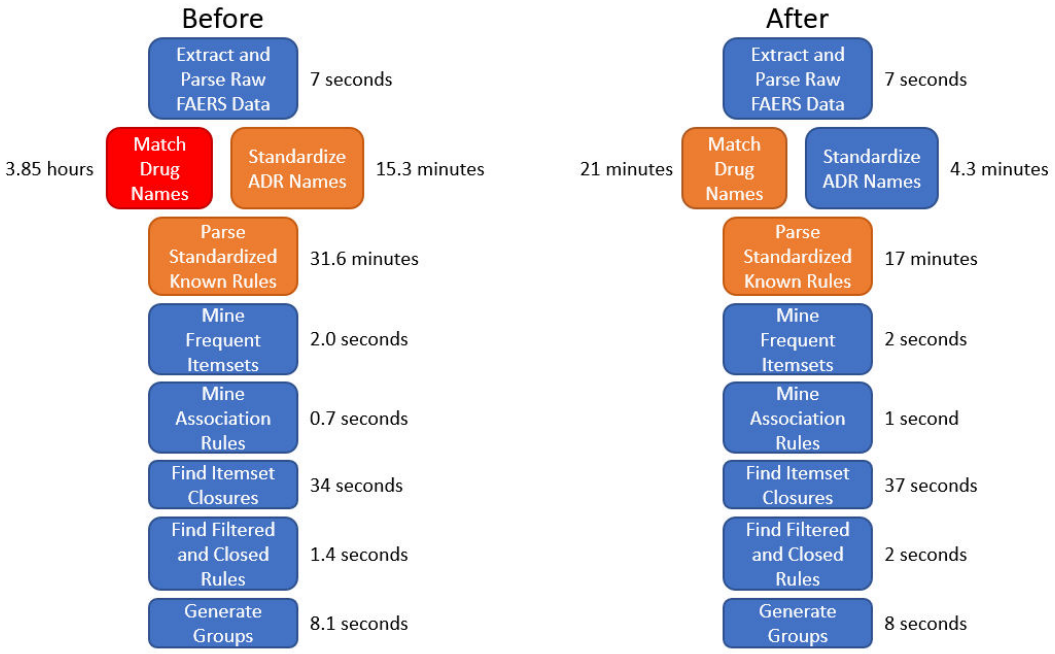


Figure 6.1.: Time Analysis Before and After Optimizations

5 (being very easy). Of the nine tasks completed by participants, all but three received an average rating of 4 or higher (see Fig. 6.2). Then, of these three, tasks 3 and 6 received an average rating above 3.8. This leaves task 8 as the task with the lowest rating for ease of use at 2.79.

Task 8 involved finding reports for interactions between two specific drugs. Users often only completed the task by asking for help from the evaluator. This indicates that although the tour explains the purpose of the Report Chip Container, users quickly forget its purpose or that it is even related to reports. After the user evaluation, we added a "Reports" label to the Report Chip Container to address this issue at the suggestion of participants (see Fig. 6.4).

Another task of interest is task 3. Although it received an average ease of use rating of 3.85, this task confused some participants. Besides having the highest average time for task completion (95 seconds), as seen in Fig. 6.3, many users indicated their confusion about the term "severe ADR". This term is explained in the help menu that all users read before beginning tasks, as well as shown on the legends below the Galaxy View and the Interaction Profile View. The intent of the task was to get users to sort the galaxy view by severity and select the top drug. One possible source of this confusion could be that the legends were often not visible without scrolling down on the webpage, so they did not serve their purpose when users forgot they were there. To address this concern, we made the website's content dynamically resize to ensure that everything can be seen without scrolling. Additionally, each of the legends was refactored to give them a clear

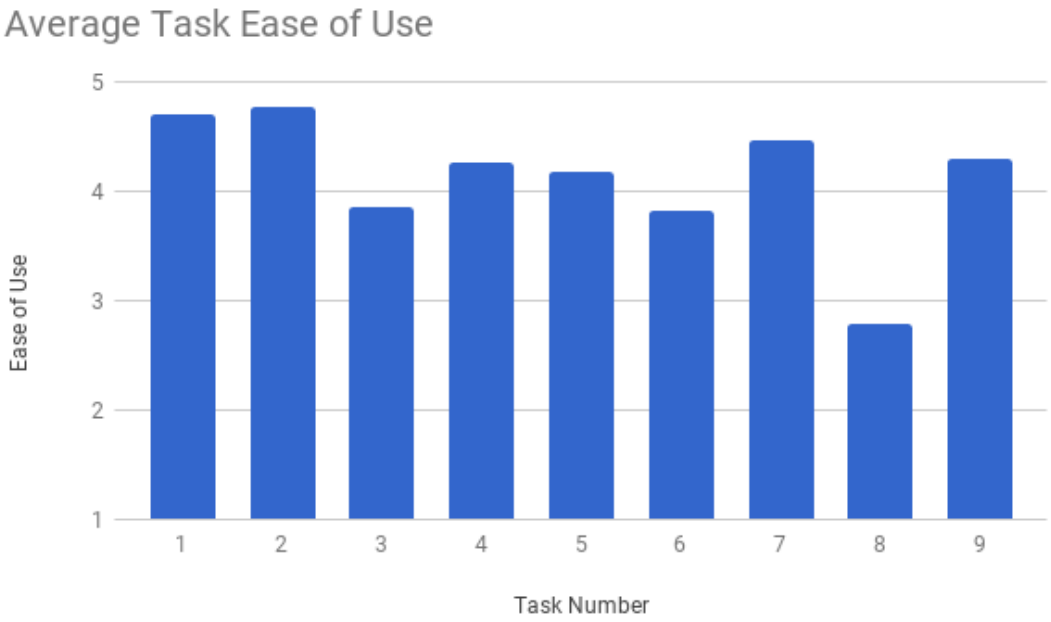


Figure 6.2.: Average Ease of Use for Each Task

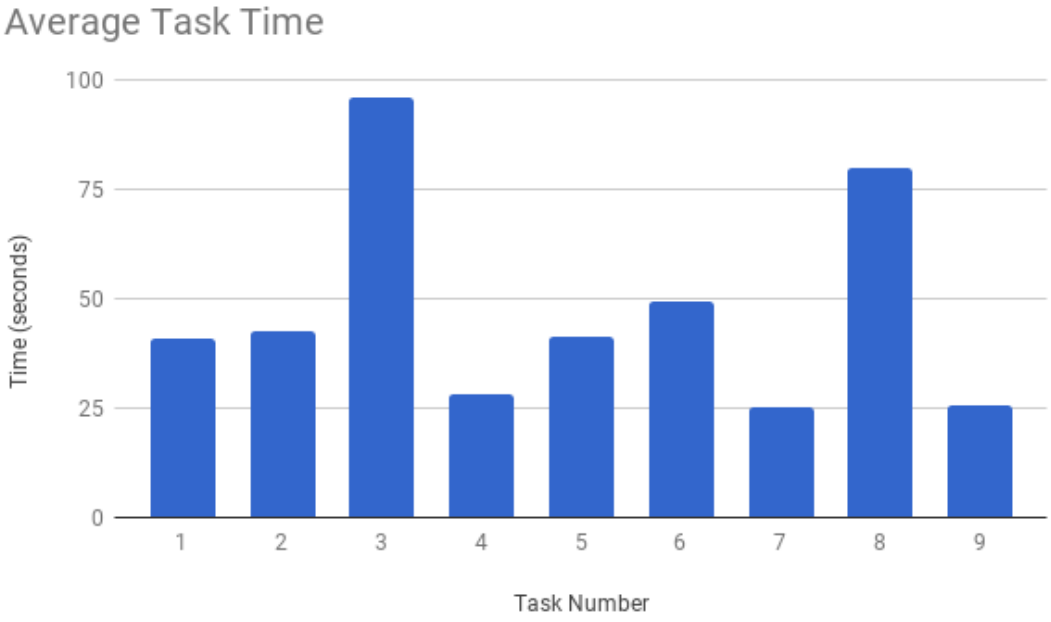


Figure 6.3.: Average Time to Complete Each Task

title that indicates what the colors represent. These changes are shown in Fig. 6.4.

Lastly, task 6 revealed an ambiguity associated with the report count for interactions between two drugs. Users received varying answers for this task based on their methodology. One way users received the answer was by going to the Galaxy View for the first drug and hovering over the edge representing the interaction with the second drug. The tooltip displayed on hovering gave an answer of 28 reports. Meanwhile, some users went to the Report View for the first drug and using the barchart received an answer of 52 reports. This discrepancy occurred because the report count given in the Galaxy View only accounted for reports between the two drugs that also included the highest-scoring ADR between those two drugs, which is represented by the edge in the Galaxy View. Meanwhile, the barchart in the Reports View for a single drug shows the number of time that a given drug appears in any report that also contains that single drug. As a result, we decided to remove the report count from the tooltip over edges in the Overview and Galaxy View and instead add the total number of ADRs to the tooltip and change the label to say that the edge represents the highest-scored ADR (see fig. 6.4).

When asked, most participants indicated that the most difficult task was either task 3 or task 8, that is the task finding the drug with the most severe ADRs or the task finding the reports for interactions between two drugs.

6.2.2. General Performance

After completing all of the tasks, users were asked to rate the general performance of the application in a variety of ways and then provide feedback in response to open-ended questions. Users rated the overall ease of use of the Overview, the Galaxy View, and the Interaction Profile View. In addition, they indicated how easy it was to remember or decide which combination of operations to use to answer a question and how easy it was to recover from errors. Of the three views, the Galaxy View was rated best for ease of use (4.35), followed by the Interaction Profile (4.11), and then the Overview (3.76).

While rated highly, there was at least one major issue discovered with the Galaxy View. When completing tasks, users often assumed that the metaphor of dashed and solid lines (for known and unknown interactions) carried over from the Overview to the Galaxy View. However, the Galaxy View currently uses node size as the method of distinguishing between known and unknown interactions, with known interactions being represented by significantly smaller nodes. Otherwise, many users found that tasks were "made simpler" by going through the Galaxy View, as it filtered out a lot of the extra information provided by the Overview and facilitated the retrieval of any information related to a given drug.

Next, there were a few complaints about the Interaction Profile. For example, some users experienced an issue where the ADRs were offscreen at the initial load of the Interaction Profile, and for this reason they did not realize you could find information about ADRs in the Interaction Profile View. Additionally, participants were confused when clicking

on a node caused parts of the tree shown in the Interaction Profile to collapse. Some particpants also observed that when a drug interacts with many other drugs, it is difficult to examine the view and locate the desired information. For this reason, they suggested that "it might be better to represent the information in a table" so that the information can be displayed in a more concise manner.

Finally, many participants were confused about why the Overview was necessary when the same information can also be found in the Galaxy and Interaction Profile views. However, this confusion is in part due to their lack of domain knowledge or experience, as the Overview is primarily for identifying highly-scored, unknown drug-drug interactions that require urgent attention. This cannot be done in the Galaxy or Interaction Profile views, as they only show information for specific drugs and do not facilitate the comparison between all interactions. Additional complaints about the Overview focused on how crowded the view is, with many interconnecting nodes in close proximity. However, for the purposes of initial screening by score and known/unknown status, this problem is mitigated using the relevant filters, and if users are looking for a specific drug, they can use the search bar.

The study showed that operation of the application takes some getting used to, as evidenced by the users' rating for ease of remembering combinations of operations (3.64). For example, some participants had trouble remembering where to click, whether it be on an edge or a node depending on the view, to achieve the desired effect. Along the same lines, users often forgot that clicking on nodes or edges produces report "chips" that can be used to find reports specific to the corresponding drugs or interactions.

Lastly, users indicated that it was fairly easy to recover from errors even if they began approaching a problem with a methodology that would not lead them to the answer, giving an average rating of 4. This is in part due to the quick response time of the application and the lack of long chains of operations being required to perform tasks within our application. If a user clicks on the wrong drug or interaction, they can simply select or search for the correct one, and the Galaxy and Interaction Profile views are updated accordingly. If a user goes to the report view and cannot find what they are looking for, they can close the view and continue analyzing the data they were looking at previously. Therefore, even if a user is still learning how to use the application, it is easy for them to recover from their mistakes and continue in pursuit of their information need.

6.2.3. Visual Appeal

The visual appeal of the application was consistently rated highly by participants (average of 4.35 out of 5). One suggestion was to change the use of dashed and solid lines such that known interactions are represented by solid lines and unknown interactions by dashed lines. Another suggestion was to make the report window larger in order to see more of the large amount of information available there. Additionally, participants suggested removing the necessity for scrolling, changing the text color in the views to make it more

visible when edges and text are overlapping, and changing the colors used for scores and severe ADR counts to make them “more distinct” and “less harsh”. The concern about removing unnecessary scrolling was addressed by making the website’s content dynamically resize to ensure that everything can be seen without scrolling. Furthermore, while the colors themselves were not changed, more variability was added by distributing the colors uniformly according to the score and severe ADR count distributions. As a result, most of the edges in the Overview that were previously a dark red color are now a milder yellow color. These changes can be seen in Fig. 6.4.

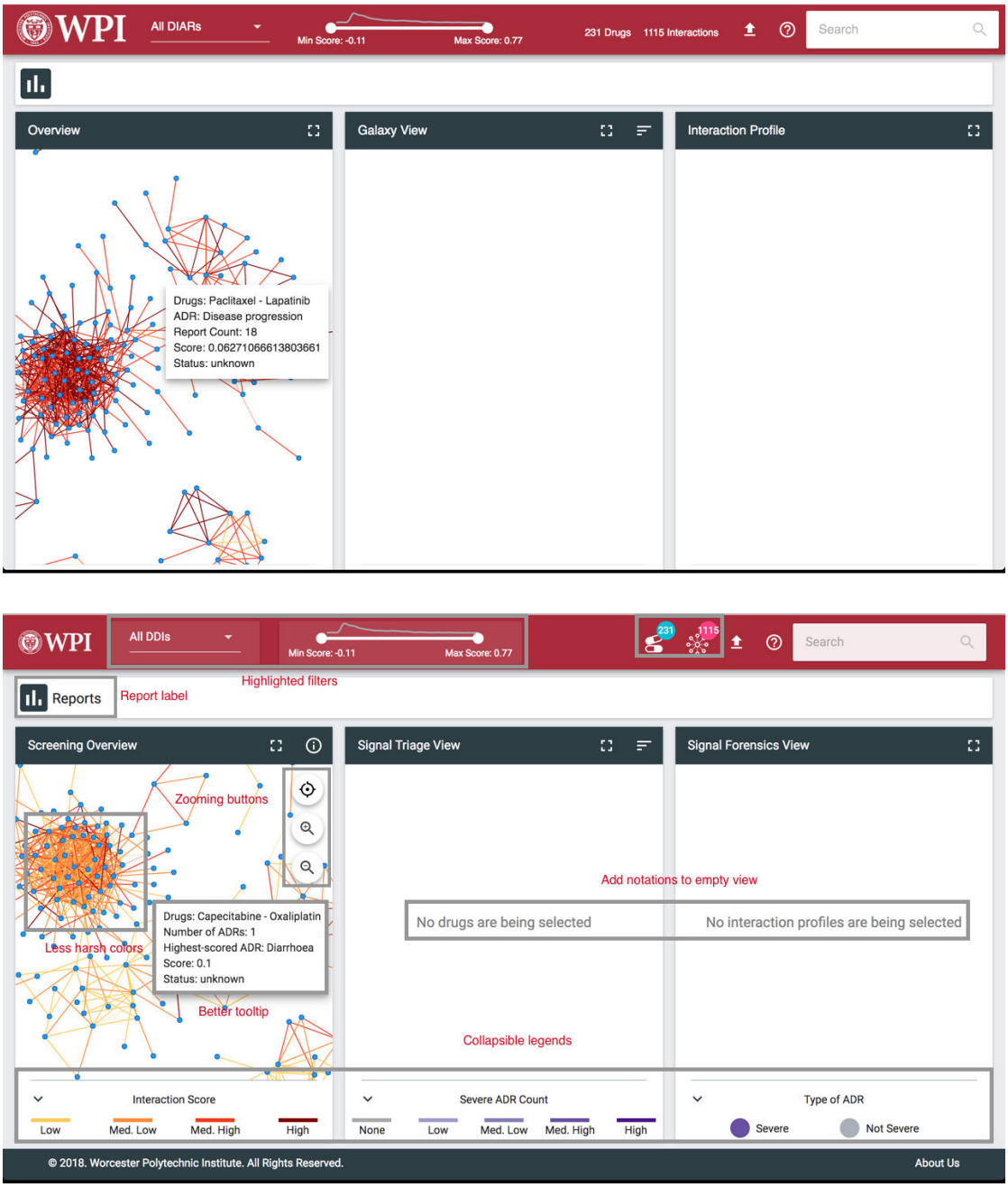


Figure 6.4.: Changes Made After the Evaluation

7. Conclusions

In this section, we summarize the accomplishments of this project and conclude by proposing areas where this project can be expanded in the future.

7.1. Summary and Findings

In this project we accomplished a complete refactor of the existing MARAS and DIVA systems into one streamlined application, MIAP. Furthermore, we enhanced both the rule-mining and data visualization capabilities of the resultant system. By making organized code repositories and extensively documenting the refactored code, we ensured that the system would be maintainable and extensible in the future. To improve the rule-mining system, we added drug matching and ADR standardization capabilities, that in turn improved the automatic labeling of known rules. When refactoring the data visualization system, we improved usability by adding features such as the score distribution coupled with the range selector for filtering scores and the generation of report "chips" that allow users to easily access reports for a specific drug or interaction. After a user evaluation, participants' feedback was incorporated by further enhancing the system to increase the system's usability. As a result of this project, users are now able to go through the entire multiple drug interaction analysis process through the user interface, from uploading FAERS files for analysis to identifying novel drug-drug interactions.

7.2. Future Work

Through the usability evaluation process, our own experience using the application, and discussions with domain specialists at the FDA, we have developed a list of recommendations for improvements that can be made to MIAP in the future. Principal among these are the integration of a database, user interface improvements for the Report View, the ability to create custom datasets, and evaluation by domain specialists with the FDA.

7.2.1. Database

A database is necessary to make this application viable because the current method of loading reports and rules takes an excessive amount of time. Currently, the reports file

is parsed each time a request for reports is made, and likewise the rules file is parsed each time a request for rules is made. As more data is added, the times for each request will only increase. By adding a database for reports and rules, waiting times could be drastically reduced when querying by report ID or drug names.

7.2.2. Report View Improvements

As is, the Report View provides users with a wealth of information in a large table that is hard to analyze. To improve the usability of this view, we suggest adding the ability to search through reports, filter reports, and change the format of the information so that it is easier to read and understand. Additionally, annotation functionality should be added to satisfy Requirement 7, meaning that domain specialists should be able to leave comments on specific reports or label rules as known, unknown, a co-occurrence, etc.

7.2.3. Visualization of Custom Datasets

In the MIAP system, users can see the datasets that have already been added and upload FAERS data through the user interface. When a new dataset is uploaded, the data analysis is ran on that dataset as well as all previously uploaded datasets to generate an aggregated visualization file. However, some users may wish to look at some subset of the datasets. For this reason we suggest adding functionality for the creation of custom datasets and investigation into methods of optimizing collective data analysis to reduce the time taken to produce a new visualization when new data is added or a custom dataset is created.

7.2.4. FDA Evaluation

Lastly, we recommend having domain specialists with the FDA evaluate MIAP to ensure that the application meets their needs and to receive any suggestions for making it easier to use. A proposed methodology for this evaluation can be found in Sec. 5.2. Specifically, we would also like to determine whether metaphors currently employed in the visualization are intuitive, such as the use of dashed lines for known rules and solid lines for unknown rules or the use of the term “edge” rather than “link”. In the case of dashed and solid lines, many users that evaluated MIAP found it counterintuitive for dashed lines to be known and solid lines to be unknown. However, we believe that given the domain, this metaphor makes sense because unknown rules are more important to FDA safety analysts and therefore are represented by solid lines.

Meanwhile, for the terminology used to describe the lines between two nodes, it is possible that FDA personnel may not be familiar with the term edge, as this term is commonly related to graph theory, a topic commonly studied in computer science but not in other fields. Therefore, these evaluations will help establish how well the system meets the

FDA's needs, as well as reveal what changes in metaphors and terminology would improve the system's usability.

8. References

"About the Center for Drug Evaluation and Research." 2018. U.S. Food and Drug Administration; Online. https://www.fda.gov/AboutFDA/CentersOffices/OfficeofMedicalProductsandTobacco/CDER/default.htm.

Aronson, Alan R, and François-Michel Lang. 2010. "An overview of MetaMap: historical perspective and recent advances." doi:10.1136/jamia.2009.002733.

"Chai Assertion Library." n.d. *Chai*. http://www.chaijs.com/.

"Drug Interactions API." 2017. Accessed December 12. https://rxnav.nlm.nih.gov/InteractionAPIs.html{\#}

Du, Mengmeng. 2005. "Approximate Name Matching." *NADA, Numerisk Analys Och Datalogi, KTH, Kungliga Tekniska Högskolan. Stockholm: Un*, 3–15.

"Express - Node.js Web Application Framework." 2017. *Express - Node.js Web Application Framework*. https://expressjs.com/.

Fournier-Viger, Philippe, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Cheng-Wei Wu, and Vincent S Tseng. 2014. "SPMF: A Java Open-Source Pattern Mining Library." *The Journal of Machine Learning Research* 15 (1). JMLR. org: 3389–93.

"Get Typescript." 2017. *TypeScript - JavaScript That Scales.* http://www.typescriptlang.org/.

Gollapudi, Sunila. 2016. *Practical Machine Learning*. Packt Publishing.

Kakar, Tabassum. 2016. "MARAS: Multi-Drug Averse Reactions Analytic System." Master's thesis, Worcester Polytechnic Institute.

Kakar, Tabassum, and Xiao Qin. 2017–2018. "In Submission to Vast 2018." Worcester Polytechnic Institute.

Kuhn, Michael, Ivica Letunic, Lars Juhl Jensen, and Peer Bork. 2016. "The SIDER database of drugs and side effects." *Nucleic Acids Research* 44 (D1): D1075–D1079. doi:10.1093/nar/gkv1075.

Li, Cong Rui Ji, and Zhi Hong Deng. 2007. "Mining frequent ordered patterns without candidate generation." *Proceedings - Fourth International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2007* 1: 402–6. doi:10.1109/FSKD.2007.402.

Liu, Simon, Wei Ma, Robin Moore, Vikraman Ganesan, and Stuart Nelson. 2005. "RxNorm: Prescription for Electronic Drug Information Exchange." *IT Professional* 7

(5). IEEE: 17–23.

Lloyd, Allison. 1999. “Dynamic Programming Algorithm (Dpa) for Edit-Distance.” http://users.monash.edu/~lloyd/tildeAlgDS/Dynamic/Edit/.

“MedDRA Hierarchy - How to Use.” 2017. Accessed December 12. https://www.meddra.org/how-to-use/basics/hierarchy.

Münch, Jürgen, Fabian Fagerholm, Patrik Johnson, Janne Pirttilahti, Juha Torkkel, and Janne Jäarvinen. 2013. “Creating Minimum Viable Products in Industry-Academia Collaborations.” In *Lean Enterprise Software and Systems*, edited by Brian Fitzgerald, Kieran Conboy, Ken Power, Ricardo Valerdi, Lorraine Morgan, and Klaas-Jan Stol, 137–51. Berlin, Heidelberg: Springer Berlin Heidelberg.

“Npm.” n.d. *Npm*. https://www.npmjs.com/.

“Preventable Adverse Drug Reactions: A Focus on Drug Interactions.” 2016. U.S. Food and Drug Administration; Online. https://www.fda.gov/drugs/developmentapprovalprocess/developmentresources/druginteractionslabeling/ucm110632.htm.

“Questions and Answers on FDA’s Adverse Event Reporting System (FAERS).” 2017. Center for Drug Evaluation; Research. https://www.fda.gov/Drugs/GuidanceComplianceRegulatoryInformation/Surveillance/AdverseDrugEffects/.

“Questions and Answers on Fda’s Adverse Event Reporting System (Faers).” 2017. U.S. Food and Drug Administration; Online. https://www.fda.gov/Drugs/GuidanceComplianceRegulatoryInformation/Surveillance/AdverseDrugEffects/default.htm.

Tatonetti, N. P., P. P. Ye, R. Daneshjou, and R. B. Altman. 2012. “Data-Driven Prediction of Drug Effects and Interactions.” *Science Translational Medicine* 4 (125): 125ra31–125ra31. doi:10.1126/scitranslmed.3003377.

“The Fun, Simple, Flexible Javascript Test Framework.” 2018. *Mocha*. https://mochajs.org/.

“Unified Medical Language System (UMLS).” 2017. U.S. National Library of Medicine. Accessed December 12. https://www.nlm.nih.gov/research/umls/.

Wishart, David S, Craig Knox, An Chi Guo, Savita Shrivastava, Murtaza Hassanali, Paul Stothard, Zhan Chang, and Jennifer Woolsey. 2006. “DrugBank: A Comprehensive Resource for in Silico Drug Discovery and Exploration.” *Nucleic Acids Research* 34 (suppl_1). Oxford University Press: D668–D672.

2018. *Node.js*. https://nodejs.org/en/.

A. Usability Evaluation Materials

A.1. Evaluation Script for WPI Student Surveys

- [Introduction: Who we are, what our project is, what is DIVA?]
  - We are [names], and we are an MQP team working on the DIVA system. DIVA is a web application that visualizes drug interaction data so safety evaluators can more easily identify and investigate potential Adverse Drug Reactions, or ADRs. ADRs are unwanted side effects that are caused by taking one or more drugs; for example, taking Aspirin and Warfarin together can cause excessive bleeding. It can be especially difficult to discover interactions between multiple drugs in clinical trials, because it's impossible to test every drug combination.
  - Our system uses data mining with adverse reaction data collected by the FDA to find likely ADRs caused by the interaction of multiple drugs. The DIVA platform allows users to examine the results of that analysis in a variety of ways. The potential ADRs found by the DIVA system are scored based on how likely or significant the interaction is.
- Do you have any questions so far?
- [Interface: Introduce each View]
  - Now we're going to introduce you to the DIVA interface. Clicking on the question mark in the top-right corner will bring up a help menu that will give you a description of the system and walk you through the interface. Please open that help menu and read the contents now.
  - [Once they are done] Now we're going to ask you to do a few tasks using the interface. Take as long as you need to complete each task. As you are working, tell us what you are thinking or feeling, what you're confused by, and any other comments you have. If, at any point, you feel that you are unable to complete the task, just let us know and we can move on to the next one. Are you ready to start?
- Data we will record during each task
  - **Time elapsed** while a participant is completing the task
  - **Answer(s)** given by participant in response to the task

  - The **methodology** that the participant used to complete the task
  - **Level of guidance/assistance** requested from the participant while completing the task
  - **Comments** from participants that indicate confusion or provide feedback as to the quality of the interface/system
- [Tasks] - after each task, the participant will be asked to rate that task's difficulty on a scale of 1 to 5, where 1 is very easy and 5 is very hard
  - Overview [**R1**]
    - **Task 1 [R3, R4]:** Find 3 unknown drug-drug interactions.
    - **Task 2 [R2]:** Find 3 drug-drug interactions with a score higher than [0.5]
  - Galaxy View
    - **Task 3 [R2, R5]:** Find and select drugs Trazodone, Metaxalone, and Tylenol. Of these drugs, which drug has the highest number of severe ADRs.
    - **Task 4 [R2, R3, R4]:** For drug Metaxalone, how many high scored [score > 0.2] and unknown interactions are there?
  - Interaction Profile
    - **Task 5 [R2]:** Please find what ADRs occur between drugs Trazodone and Diazepam.
    - **Task 6 [R6]:** How many reports support the above interaction?
    - **Task 7 [R2, R5]:** Find all severe ADRs associated with Trazodone's interactions.
  - Report View [**R6**]
    - **Task 8:** Please open the reports for the interaction between drugs Zoloft and Tylenol.
    - **Task 9:** List the three most frequent drugs reported with the interacting drugs Zoloft and Tylenol.
- Multiple Choice Questions
  - On a scale of 1 to 5, how easy was it to:
    - **Question 1:** Interpret the information contained in the overview
    - **Question 2:** Interpret the information contained in the galaxy view
    - **Question 3:** Interpret the information contained in the profile view

        * **Question 4:** Remember/decide which combination of operations to use to answer a question
        * **Question 5:** Recover from errors once I realized a given approach would not lead me to the information I needed.
    - On a scale of 1 to 5, how visually appealing is the interface, where 1 is not appealing at all and 5 is extremely appealing?
- [Open-ended Questions]
    - Which task was the most difficult to complete? What made it so difficult?
    - Which parts of the interface are the most difficult to understand? Why?
    - Which parts of the interface are the most difficult to use? Why?
    - Is there any way you could think of to make the interface easier to understand?
    - Is there any way you could think of to make the interface faster to use?
    - What would you change, if anything, to make the interface more visually appealing?

A.2. Evaluation Script for FDA Researchers

- [Introduction: Who we are, what our project is, what DIVA is]
    - We are [names], and we are an MQP team working on the DIVA system. DIVA is a web application that visualizes drug interaction data so FDA drug safety evaluators can more easily identify and investigate potential Adverse Drug Reactions, or ADRs.
- [Preliminary Questions]
    - What is your role at the FDA?
    - How do you work with ADRs and FAERS data?
- [Introduction to DIVA platform]
    - Our system uses data mining with data collected by FAERS to find likely ADRs caused by the interaction of multiple drugs. The DIVA platform allows users to examine the results of that analysis in a variety of ways. The potential ADRs found by the DIVA system are ranked by their contrast score. This essentially weighs how likely it is that a reaction is a result of taking multiple drugs in combination versus taking one of the drugs individually.. For example, in the case of Aspirin and Warfarin, when someone takes both of those drugs together, there is a high chance that they will experience excessive bleeding; however, when they take either drug on their own, the chance of them experiencing

that reaction is relatively low. This indicates that the interaction between Aspirin and Warfarin likely causes excessive bleeding, and so that interaction would have a high contrast score.

- [General Questions]
  - How would you use a tool such as the one described above? What are the main features you would look for in a similar tool?
- [Show them the Interface]
  - Here is the DIVA Interface. Clicking on the question mark in the top-right corner will give you a brief description of how to use the platform. Please open the help menu and read it now.
  - Do you have any questions about the interface?
  - Thinking back to how you said you would use such a tool, try to use the interface in each of those ways.
  - On a scale of 1 to 10, how useful do you think this interface would be for your day-to-day job, where 1 is not useful at all and 10 is invaluable?
  - What feature or features do you think are most useful to you?
  - What feature or features do you think would be least useful to you, if any?
  - What changes or improvements would you make to this interface to make it more useful to you as an FDA employee?
  - On a scale of 1 to 10, how difficult is this interface to understand, where 1 is not difficult at all and 10 is extremely difficult?
  - Do you have any additional questions or comments for us?

A.3. Usability Evaluation Additional Results

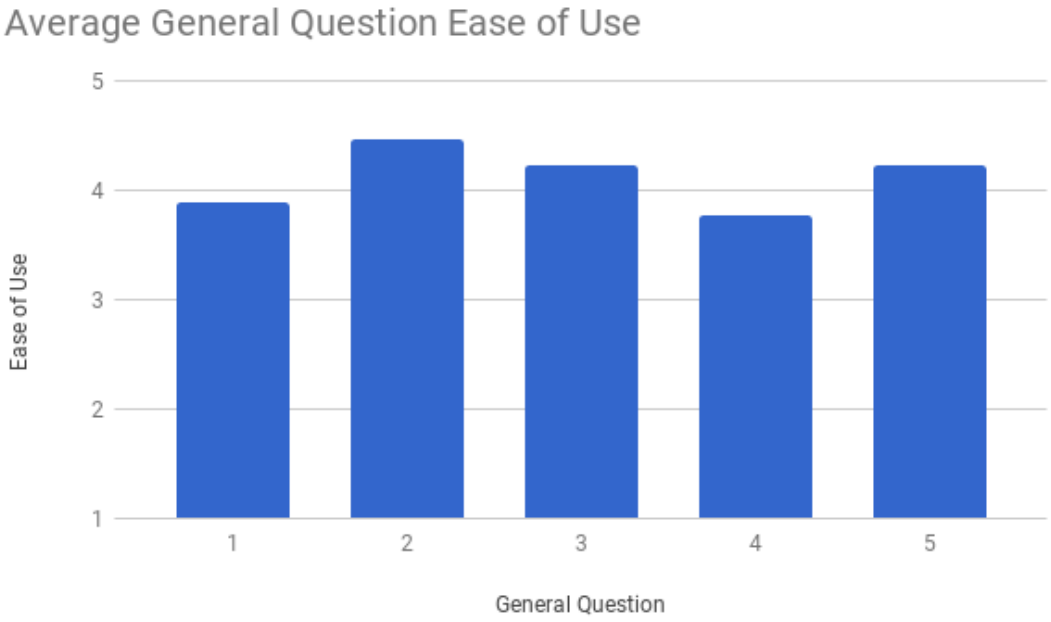


Figure A.1.: Average Ease of Use for Each General Question

B. User Interface (React.js) Documentation

`resources\assets\js\components\App.jsx` App This is the main component for the application. It contains the Tour, GlobalFilterNavContainer, MainviewContainer, and Footer components.

`resources\assets\js\components\layouts\DndTreeContainer.jsx` DndTreeContainer This component creates the windows for each of the current drugs displayed in the Galaxy View.

Table B.1.: DndTreeContainer

Property	Type	Required	Description
currentDrugs	PropType array	yes	Array of drugs currently in the Galaxy View.
selectedDrug	PropType string	yes	Name of the currently selected drug.
scoreRange	PropType array	yes	Array of score boundaries, indicating how to color nodes/edges based on score.
dmeRange	PropType array	yes	Array of severe ADR count boundaries, indicating how to color galaxy view headers.
filter	PropType string		Can be 'all', 'known', or 'unkown'. Corresponds to filtering interactions by known/unknown.

Property	Type	Required	Description
minScore	PropType number		Minimum score for filtering interactions.
maxScore	PropType number		Maximum score for filtering interactions.
onClickNode	PropType func		Callback used when a node is clicked. Takes the node as a parameter.
onClickEdge	PropType func		Callback used when a edge is clicked. Takes the edge as a parameter.
onDeleteNode	PropType func		Callback used when a drug is removed from the galaxy view. Takes the drug name as a parameter.
onClearDrug	PropType func		Clears the selectedDrug.
cols	PropType number		Number of columns currently being displayed in Mainview (4 if all views visible or 12 if one is fullscreened).
handleOpen	PropType func		Used to open the reports view. Takes a report object containing information about the drug for which to retrieve reports.

Property	Type	Required	Description
helpExample	PropType bool		Indicates whether this is the version found in the help menu (defaults to false).

`resources\assets\js\components\layouts\GlobalFilterNav.jsx` **GlobalFilterNav** This component renders the navbar, combining the filters, upload button, help button, and search bar.

Table B.2.: GlobalFilterNav

Property	Type	Required	Description
rules	PropType array	yes	Array of rules representing all interaction between pairs of drugs in the visualization.
updating	PropType bool	yes	Indicates whether filters are currently being applied.
scoreRange	PropType array	yes	Array of score boundaries, indicating how to color nodes/edges based on score.
dmeRange	PropType array	yes	Array of severe ADR count boundaries, indicating how to color galaxy view headers.
minScore	PropType number	yes	Minimum score for filtering interactions.
maxScore	PropType number	yes	Maximum score for filtering interactions.

Property	Type	Required	Description
filter	PropType string	yes	Can be ‘all’, ‘known’, or ‘unkown’. Corresponds to filtering interactions by known/unknown.
status	PropType object	yes	Contains information about the status of the last MARAS analysis ran.
onClick	PropType func	yes	Used to apply the known/unknown filter. Takes the value of the selected option (‘all’, ‘known’, or ‘unknown’).
updateMinScore	PropType func	yes	Used to set the new minimum score. Takes the new minimum score (a number) as a paramter.
updateMaxScore	PropType func	yes	Used to set the new maximum score. Takes the new maximum score (a number) as a paramter.
isUpdating	PropType func	yes	Used to indicate that the visualization is updating as a new filter has been applied. Takes a boolean indicating whether updating is in progress.

Property	Type	Required	Description
getStatus	PropType func	yes	A function that can be called to get the updated analysis status.

`resources\assets\js\components\layouts\MainView.jsx` **MainView** This component combines the ReportChipContainer, Overview, Galaxy View, Interaction Profile and Report View.

Table B.3.: MainView

Property	Type	Required	Description
isFetching	PropType bool	yes	Indicates whether the data is still being fetched for the nodes and links.
links	PropType array	yes	Array of links representing all interaction between pairs of drugs in the visualization.
nodes	PropType array	yes	Array of nodes representing all drugs in the visualization.
currentDrugs	PropType array	yes	Array of drugs currently in the Galaxy View.
selectedDrug	PropType string	yes	Name of the currently selected drug.
selectedRule	PropType string	yes	Name of the currently selected rule (of format: drug_1 — drug_2).

Property	Type	Required	Description
filter	PropType string	yes	Can be ‘all’, ‘known’, or ‘unkown’. Corresponds to filtering interactions by known/unknown.
minScore	PropType number	yes	Minimum score for filtering interactions.
maxScore	PropType number	yes	Maximum score for filtering interactions.
scoreRange	PropType array	yes	Array of score boundaries, indicating how to color nodes/edges based on score.
dmeRange	PropType array	yes	Array of severe ADR count boundaries, indicating how to color galaxy view headers.
status	PropType object	yes	Contains information about the status of the last MARAS analysis ran.
onClickNode	PropType func	yes	Callback used when a node is clicked. Takes the node as a parameter.
onClickEdge	PropType func	yes	Callback used when an edge is clicked. Takes the edge as a parameter.

Property	Type	Required	Description
showDetailNode	PropType func	yes	Callback used when a node is clicked (Galaxy View). Takes the node as a parameter.
deleteNode	PropType func	yes	Callback used when a drug is removed from the galaxy view. Takes the drug name as a parameter.
isUpdating	PropType func	yes	Used to indicate that the visualization is updating as a new filter has been applied. Takes a boolean indicating whether updating is in progress.
clearRule	PropType func	yes	Remove the currently selected rule from the Interaction Profile.
clearSearchTerm	PropType func	yes	Remove the currently selected drug from the Interaction Profile.
getStatus	PropType func	yes	Used to get updated information about the status of the last MARAS analysis ran.

`resources\assets\js\components\modules\D3Tree.jsx` **D3Tree** This component renders the tree seen in the Profile View. This component should not be used directly. Instead, use InteractionProfile.

Table B.4.: D3Tree

Property	Type	Required	Description
scoreRange	PropType array	yes	Array of score boundaries, indicating how to color nodes/edges based on score.
treeData	PropType array	yes	Information about how to structure the tree.
filter	PropType string	yes	Can be 'all', 'known', or 'unkown'. Corresponds to filtering interactions by known/unknown.
minScore	PropType number	yes	Minimum score for filtering interactions.
maxScore	PropType number	yes	Maximum score for filtering interactions.
width	PropType number	yes	Width of the parent node.
height	PropType number	yes	Height of the parent node.

`resources\assets\js\components\modules\DistributionRangeSlider.jsx` **DistributionRangeSlider** This component shows a distribution of the scores of all links over a range slider used to filter by score.

Table B.5.: DistributionRangeSlider

Property	Type	Required	Description
rules	PropType array	yes	Array of rules representing all interaction between pairs of drugs in the visualization.

Property	Type	Required	Description
updateMinScore	PropType func		Used to set the new minimum score. Takes the new minimum score (a number) as a paramter.
updateMaxScore	PropType func		Used to set the new maximum score. Takes the new maximum score (a number) as a paramter.
isUpdating	PropType func		Used to indicate that the visualization is updating as a new filter has been applied. Takes a boolean indicating whether updating is in progress.
helpExample	PropType bool		Indicates whether this is the version found in the help menu (defaults to false).

`resources\assets\js\components\modules\DndGraph.jsx` **DndGraph** This component renders the graph seen in the Overview.

Table B.6.: DndGraph

Property	Type	Required	Description
nodes	PropType array	yes	Array of nodes representing all drugs in the visualization.

Property	Type	Required	Description
links	PropType array	yes	Array of links representing all interaction between pairs of drugs in the visualization.
width	PropType number	yes	Width of the browser window.
height	PropType number	yes	Height of the browser window.
selectedDrug	PropType string		Name of the currently selected drug.
onClickNode	PropType func		Callback used when a node is clicked. Takes the node as a parameter.
onClickEdge	PropType func		Callback used when an edge is clicked. Takes the edge as a parameter.
isFetching	PropType bool		Indicates whether the data is still being fetched for the nodes and links.
filter	PropType string		Can be 'all', 'known', or 'unkown'. Corresponds to filtering interactions by known/unknown.
minScore	PropType number		Minimum score for filtering interactions.
maxScore	PropType number		Maximum score for filtering interactions.

Property	Type	Required	Description
isUpdating	PropType func		Used to indicate that the visualization is updating as a new filter has been applied. Takes a boolean indicating whether updating is in progress.
scoreRange	PropType array	yes	Array of score boundaries, indicating how to color nodes/edges based on score.

`resources\assets\js\components\modules\DndTree.jsx` **DndTree** This component controls and renders a single galaxy view inside of a DndTreeContainer.

Table B.7.: DndTree

Property	Type	Required	Description
scoreRange	PropType array	yes	Array of score boundaries, indicating how to color nodes/edges based on score.
helpExample	PropType bool		Indicates whether this is the version found in the help menu (defaults to false).
currentDrug	PropType string	yes	Name of the central drug.
data	PropType object	yes	Contains information such as rules that allow for rendering the graph.

Property	Type	Required	Description
filter	PropType string		Can be 'all', 'known', or 'unkown'. Corresponds to filtering interactions by known/unknown.
minScore	PropType number		Minimum score for filtering interactions.
maxScore	PropType number		Maximum score for filtering interactions.
onClickEdge	PropType func		Callback used when an edge is clicked. Takes the edge as a parameter.

`resources\assets\js\components\modules\Footer.jsx` **Footer** This component defines the footer shown fixed at the bottom of the page.

`resources\assets\js\components\modules\GalaxyView.jsx` **GalaxyView** This component renders the Galaxy View.

Table B.8.: GalaxyView

Property	Type	Required	Description
col	PropType number	yes	Number of columns currently being displayed in Mainview (4 if all views visible or 12 if one is fullscreened).
toggleFullscreenGalaxy	PropType func	yes	Function that fullscreens the Galaxy View.
currentDrugs	PropType array	yes	Array of drugs currently in the Galaxy View.

Property	Type	Required	Description
filter	PropType string	yes	Can be ‘all’, ‘known’, or ‘unkown’. Corresponds to filtering interactions by known/unknown.
minScore	PropType number	yes	Minimum score for filtering interactions.
maxScore	PropType number	yes	Maximum score for filtering interactions.
width	PropType number	yes	Width of the browser window.
height	PropType number	yes	Height of the browser window.
onClickNode	PropType func	yes	Callback used when a node is clicked. Takes the node as a parameter.
onClickEdge	PropType func	yes	Callback used when a edge is clicked. Takes the edge as a parameter.
onDeleteNode	PropType func	yes	Callback used when a drug is removed from the galaxy view. Takes the drug name as a parameter.
onClearDrug	PropType func	yes	Clears the selectedDrug.
handleOpen	PropType func	yes	Used to open the reports view. Takes a report object containing information about the drug for which to retrieve reports.

Property	Type	Required	Description
scoreRange	PropType array	yes	Array of score boundaries, indicating how to color nodes/edges based on score.
dmeRange	PropType array	yes	Array of severe ADR count boundaries, indicating how to color galaxy view headers.
isGalaxyFullscreen	PropType bool	yes	Indicates whether the Galaxy View is currently fullscreened.
selectedDrug	PropType string	yes	Name of the currently selected drug.

`resources\assets\js\components\modules\Help.jsx` **Help** This component renders and controls the help menu.

Table B.9.: Help

Property	Type	Required	Description
scoreRange	PropType array	yes	Array of score boundaries, indicating how to color nodes/edges based on score.
dmeRange	PropType array	yes	Array of severe ADR count boundaries, indicating how to color galaxy view headers.
startTour	PropType func	yes	Used to start the tour.
rules	PropType array	yes	Array of all rules in the visualization

`resources\assets\js\components\modules\InteractionProfile.jsx` **InteractionProfile** This component is a wrapper for the D3Tree component. It controls what nodes/links are passed to the D3Tree, applying all filters.

Table B.10.: InteractionProfile

Property	Type	Required	Description
helpExample	PropType bool		Indicates whether this is the version found in the help menu (defaults to false).
scoreRange	PropType array	yes	Array of score boundaries, indicating how to color nodes/edges based on score.
mainDrug	PropType string		Name of the currently selected drug.
mainRule	PropType string		Name of the currently selected rule (of format: drug_1 — drug_2).
filter	PropType string		Can be 'all', 'known', or 'unkown'. Corresponds to filtering interactions by known/unknown.
minScore	PropType number		Minimum score for filtering interactions.
maxScore	PropType number		Maximum score for filtering interactions.

`resources\assets\js\components\modules\KnownUnknownDropDown.jsx` **KnownUnknownDropDown** This component controls and renders the filter for selecting all rules, only known rules, or only unknown rules.

Table B.11.: KnownUnknownDropDown

Property	Type	Required	Description
isUpdating	PropType func	yes	Used to indicate that the visualization is updating as a new filter has been applied. Takes a boolean indicating whether updating is in progress.
onClick	PropType func	yes	Used to apply the known/unknown filter. Takes the value of the selected option (‘all’, ‘known’, or ‘unknown’).

`resources\assets\js\components\modules\Overview.jsx` **Overview** This component controls and renders the Overview.

Table B.12.: Overview

Property	Type	Required	Description
col	PropType number	yes	Number of columns currently being displayed in Mainview (4 if all views visible or 12 if one is fullscreened).
toggleFullscreenOverview	PropType func	yes	Function that fullscreens the Overview.
onClickNode	PropType func	yes	Callback used when a node is clicked. Takes the node as a parameter.

Property	Type	Required	Description
onClickEdge	PropType func	yes	Callback used when an edge is clicked. Takes the edge as a parameter.
currentSelector	PropType string	yes	Class name of the component the tour is currently looking at.
nodes	PropType array	yes	Array of nodes representing all drugs in the visualization.
links	PropType array	yes	Array of links representing all interaction between pairs of drugs in the visualization.
width	PropType number	yes	Width of the browser window.
height	PropType number	yes	Height of the browser window.
selectedDrug	PropType string	yes	Name of the currently selected drug.
isFetching	PropType bool	yes	Indicates whether the data is still being fetched for the nodes and links.
filter	PropType string	yes	Can be 'all', 'known', or 'unkown'. Corresponds to filtering interactions by known/unknown.
minScore	PropType number	yes	Minimum score for filtering interactions.

Property	Type	Required	Description
maxScore	PropType number	yes	Maximum score for filtering interactions.
isUpdating	PropType func	yes	Used to indicate that the visualization is updating as a new filter has been applied. Takes a boolean indicating whether updating is in progress.
scoreRange	PropType array	yes	Array of score boundaries, indicating how to color nodes/edges based on score.
nextTourStep	PropType func	yes	Advances the tour to the next step.
isOverviewFullscreen	PropType bool	yes	Indicates whether the Overview is currently fullscreened.
status	PropType object	yes	Contains information about the status of the last MARAS analysis ran.
getStatus	PropType func	yes	Used to get updated information about the status of the last MARAS analysis ran.

`resources\assets\js\components\modules\ProfileView.jsx` **ProfileView** This component is used to render the Interaction Profile View.

Table B.13.: ProfileView

Property	Type	Required	Description
col	PropType number	yes	Number of columns currently being displayed in Mainview (4 if all views visible or 12 if one is fullscreened).
toggleFullscreenProfile	PropType func	yes	Function that fullscreens the Profile View.
selectedDrug	PropType string	yes	Name of the currently selected drug.
selectedRule	PropType string	yes	Name of the currently selected rule (of format: drug_1 — drug_2).
scoreRange	PropType array	yes	Array of score boundaries, indicating how to color nodes/edges based on score.
filter	PropType string	yes	Can be 'all', 'known', or 'unkown'. Corresponds to filtering interactions by known/unknown.
minScore	PropType number	yes	Minimum score for filtering interactions.
maxScore	PropType number	yes	Maximum score for filtering interactions.
isProfileFullscreen	PropType bool	yes	Indicates whether this view is fullscreened or not.

Property	Type	Required	Description
profileTitle	PropType string	yes	Used as the title for this view.

`resources\assets\js\components\modules\Report.jsx` **Report** This component shows all reports for a given drug or interaction in a Dialog box.

Table B.14.: Report

Property	Type	Required	Description
tableData	PropType array	yes	Array of the reports to display.
drugs	PropType array	yes	Array of the drug (if there is only one) or drugs (if the reports are for a pair of drugs) pertaining to this report.
windowWidth	PropType number	yes	Width of the browser window (used to make components responsive).
open	PropType bool	yes	Indicates whether the report view should be open or not.
handleClose	PropType func	yes	Called when the report view should be closed.
tableTitle	PropType string	yes	Title for the report view.

`resources\assets\js\components\modules\ReportChipContainer.jsx` **ReportChipContainer** This component holds chips that link to the report view for any drug or interaction that has been selected.

Table B.15.: ReportChipContainer

Property	Type	Required	Description
currentDrugs	PropType array	yes	Array of the currently selected drugs (i.e. the drugs found in the Galaxy View)
links	PropType array	yes	Array of the links between all drugs that interact with eachother.
selectedDrug	PropType string	yes	Name of the currently selected drug.
selectedRule	PropType string	yes	Name of the currently selected rule.
deleteNode	PropType func	yes	Removes the drug from the Galaxy View.
clearSearchTerm	PropType func	yes	Remove the currently selected drug from the Interaction Profile.
clearRule	PropType func	yes	Remove the currently selected rule from the Interaction Profile.
scoreRange	PropType array	yes	Distribution of scores used to generate color.
handleOpen	PropType func	yes	Open reports for a given drug or interaction.

`resources\assets\js\components\modules\SearchBarComponent.jsx` **SearchBarComponent** This component allows users to search for drugs in the context of this application.

Table B.16.: SearchBarComponent

Property	Type	Required	Description
drugs	PropType array	yes	Array of all drug names to be used for AutoComplete.
handleSearchRequest	PropType func	yes	Function to handle the request for a given drug when it is searched. Takes the selected drug name as a parameter.

`resources\assets\js\components\modules\StatusInformation.jsx` **StatusInformation** This component shows status information about the last MARAS analysis ran.

StatusInformationButton This component is an icon button that shows a dialog with status information about the last MARAS run when pressed.

Table B.17.: StatusInformationButton

Property	Type	Required	Description
status	PropType object	yes	The status of the last analysis ran.
getStatus	PropType func	yes	A function that can be called to get the updated analysis status.

`resources\assets\js\components\modules\Tour.jsx` **Tour** This component contains all of the tour steps and controls operation of the tour.

See https://github.com/gilbarbara/react-joyride

Table B.18.: Tour

Property	Type	Required	Description
updateTourSelector	PropType func	yes	Callback used when the selector the tour is pointing to is changed. Takes the classname of the currently selected component as a parameter.
stopTour	PropType func	yes	Function that can be called to stop the tour.
tourRunning	PropType bool	yes	Boolean indicating whether the tour should be running or not.

`resources\assets\js\components\modules\TreeViewFilter.jsx` **TreeViewFilter**

This component is used to provide users with the option of sorting the drugs in the galaxy view by various properties.

Table B.19.: TreeViewFilter

Property	Type	Required	Description
onClickRadio	PropType func	yes	Function that can be called to sort the drugs in the Galaxy View. Takes a string parameter indicating what to sort by.

`resources\assets\js\components\modules\UploadFAERS.jsx` **UploadFAERS**

This component is used to upload FAERS data zip files and show status information about the last analysis ran.

Table B.20.: UploadFAERS

Property	Type	Required	Description
status	PropType object	yes	The status of the last analysis ran.
getStatus	PropType func	yes	A function that can be called to get the updated analysis status.

C. User Interface (JavaScript) Documentation

C.1. Constants

loggerMiddleware Entry point of the React App

currentDrugs Fetch the drugs from the server and add to state object

Param	Type
state	
action	*

selectDrug Select a drug or remove the selected drug

Param	Type
state	string
action	

selectMaxScore Select the max score or set default to 2

Param	Type
state	number
action	*

selectMinScore Select the min score or set default to 2

Param	Type
state	number
action	*

selectRule Select a rule or remove the currently selected rule

Param	Type
state	string
action	*

treeViewSorting Select the sorting method in the galaxy view

dmeColors Colors and labels associated with different ranges of severe ADR counts

scoreColors Colors and labels associated with different score ranges

scoreBorderColors Colors associated with node borders for different score ranges (used in Interaction Profile)

baseNodeColor Color of base node in Interaction Profile (same as color of nodes in Overview).

baseNodeBorderColor Border color of base node in Interaction Profile.

severeADRColor Color of severe ADRs in Interaction Profile.

regularADRColor Color of non-severe ADRs in Interaction Profile.

barColor Color of bars in barchart of Report View.

barSelectedColor Color of selected bar in barchart of Report View.

selectedColor Color used for bar indicating the selected tab.

primaryColor Primary color of the application (used as background of toolbar among others).

secondaryColor Secondary color of the application (used by score distribution).

complementaryColor Color used for tabs, footer, etc.

wpiLogo WPI logo shown on left of toolbar.

teamPhoto Team picture found on About Us page.

medicines Drug count icon for toolbar.

connection Interaction count icon for toolbar.

overviewName Display name for the view on the left (previously Overview).

galaxyViewName Display name for the middle view (previously Galaxy View).

interactionProfileName Display name for the view on the right (previously Interaction Profile).

generateColor Generate color based on score

Param	Type
score	number
scoreRange	array

generateScoreBorderColor Border colors of interaction profile nodes based on score

Param	Type
score	number
scoreRange	array

getStyleByDMECount Get the style of the background of galaxy panels based on number of DMEs

Param	Type
numDMEs	number
dmeRange	array

countDrugInteraction Count number of drugs and interactions after applying filter

Param	Type
rules	array
filter	string
minScore	number
maxScore	number

C.2. Functions

visibilityFilter(state, action) Select the filtering method in overview

Param	Type	Default
state	string	“all”
action	*	

D. Server (TypeScript) Documentation

D.1. Index

D.1.1. External modules

- “server/App”
- “server/csv/CSVController”
- “server/csv/index”
- “server/csv/routes”
- “server/index”

D.2. External module: “server/App”

D.2.1. Index

D.2.1.1. Classes

- App

D.3. Class: App

Configure the server application.

D.3.1. Constructors

D.3.1.1. new App(): App

Defined in server/App.ts:12

Returns: App

D.3.2. Properties

D.3.2.1. express

- **express**: *express.Application*

Defined in server/App.ts:12

D.3.3. Methods

D.3.3.1. <<Private>> middleware

- **middleware**(): `void`

  Defined in server/App.ts:28

  Configure Express middleware.

  Returns: `void`

D.3.3.2. <<Private>> routes

- **routes**(): `void`

  Defined in server/App.ts:39

  Configure API endpoints.

  Returns: `void`

D.4. External module: "server/csv/CSVController"

D.4.1. Index

D.4.1.1. Classes

- CSVController

D.4.1.2. Variables

- exec

D.4.1.3. Functions

- csvToJson
- getDrugsFromRules

D.4.2. Variables

D.4.2.1. exec

- **exec**: *any* = require('child_process').exec

  Defined in server/csv/CSVController.ts:6

D.5. Class: CSVController

Controller class to handle different requests for this feature

D.5.1. Index

D.5.1.1. Methods

- getDMEs
- getReports
- getRules
- getStatus
- uploadReports

D.5.2. Methods

D.5.2.1. getDMEs

- **getDMEs**(req: *Request*, res: *Response*, next: *NextFunction*): Promise.<void>

Defined in server/csv/CSVController.ts:253

Retrieve array of severe ADR names.

Parameters:

Param	Type	Description
req	`Request`	-
res	`Response`	-
next	`NextFunction`	-

Returns: `Promise.<void>`

D.5.2.2. getReports

- **getReports**(req: *`Request`*, res: *`Response`*, next: *`NextFunction`*): `Promise.<void>`

  Defined in server/csv/CSVController.ts:164

  Retrieve array of reports for a given drug (req.query.drug) or interaction (req.query.drug1 and req.query.drug2).

  Parameters:

Param	Type	Description
req	`Request`	-
res	`Response`	-
next	`NextFunction`	-

  Returns: `Promise.<void>`

D.5.2.3. getRules

- **getRules**(req: *`Request`*, res: *`Response`*, next: *`NextFunction`*): `Promise.<void>`

  Defined in server/csv/CSVController.ts:51

  Get rules, drugs, as well as score and severe ADR count distributions in json.

  Parameters:

Param	Type	Description
req	`Request`	-
res	`Response`	-
next	`NextFunction`	-

Returns: `Promise.<void>`

D.5.2.4. getStatus

- **getStatus**(req: *`Request`*, res: *`Response`*, next: *`NextFunction`*): `Promise.<void>`

  Defined in server/csv/CSVController.ts:237

  Retrieve status information in json.

  Parameters:

Param	Type	Description
req	`Request`	-
res	`Response`	-
next	`NextFunction`	-

  Returns: `Promise.<void>`

D.5.2.5. uploadReports

- **uploadReports**(req: *`Request`*, res: *`Response`*, next: *`NextFunction`*): `Promise.<any>`

  Defined in server/csv/CSVController.ts:20

  Used to upload FAERS files to the server. Files are passed in as req.files.

  Parameters:

Param	Type	Description
req	`Request`	-
res	`Response`	-
next	`NextFunction`	-

Returns: `Promise.<any>`

D.6. External module: "server/csv/index"

D.6.1. Index

D.6.1.1. Functions

- init

D.6.2. Functions

D.6.2.1. init

- **init**(app: *express.Application*): `void`

  Defined in server/csv/index.ts:10

  Initialize the routes and other config in the future such as database configuration

  Parameters:

Param	Type	Description
app	`express.Application`	-

  Returns: `void`

D.7. External module: "server/csv/routes"

D.7.1. Index

D.7.1.1. Variables

- crypto

D.7.1.2. Functions

- default

D.7.2. Variables

D.7.2.1. <<Const>> crypto

- **crypto**: *any* = require('crypto')

  Defined in server/csv/routes.ts:4

D.7.3. Functions

D.7.3.1. default

- **default**(app: *express.Application*): `void`

  Defined in server/csv/routes.ts:4

  This is where we register the routes, route middlewares for this resource

  Parameters:

Param	Type	Description
app	`express.Application`	-

  Returns: `void`

D.8. External module: "server/index"

D.8.1. Index

D.8.1.1. Variables

- port
- server

D.8.1.2. Functions

- normalizePort
- onError
- onListening

D.8.2. Variables

D.8.2.1. <<Const>> port

- **port**: *string* | *number* | *true* | *false* = normalizePort(process.env.PORT || 3000)

  Defined in server/index.ts:11

  Set default port to 3000.

D.8.2.2. <<Const>> server

- **server**: *Server* = http.createServer(App)

  Defined in server/index.ts:17

  Configure server.

D.8.3. Functions

D.8.3.1. normalizePort

- **normalizePort**(val: *number* | *string*): `number` | `string` | `boolean`

  Defined in server/index.ts:27

  Make sure port number is valid.

  Parameters:

Param	Type	Description
val	`number`	`string`

  Returns: `number` | `string` | `boolean`

D.8.3.2. onError

- **onError**(error: *ErrnoException*): `void`

  Defined in server/index.ts:43

  Log error output.

  Parameters:

Param	Type	Description
error	`ErrnoException`	-

  Returns: `void`

D.8.3.3. onListening

- **onListening**(): `void`

  Defined in server/index.ts:63

  Print the port that the server is listening on.

  Returns: `void`

E. DIVA Web Application Documentation

E.1. Install dependencies for client and server application

Open your favorite Terminal and run the following command. Make sure you have the latest npm and node version. In the root directory of the application, run these commands to get started. `js static $ npm install $ npm install -g gulp $ npm install -g nodemon`

E.2. How to ensure the server keeps running after your session ends

Before running the server or building the front-end react app, use the following command to create a new terminal instance that will remain after you terminate your session: `js static $ screen`

E.3. How to build the front-end React app

The following commands build and bundle react app into plain js (choose one of the following). The compiled javascript file will be in public/ directory

When quickly compiling locally run this: `js static $ npm run fdev` When working locally, run this to continually update saved changes: `js static $ npm run fwatch` When deploying and NOT DEBUGGING, build it as production app as this will minify and otherwise optimize the app: `js static $ npm run fproduction`

E.4. How to run the server

To build typescript files into javascript files use `js static $ npm run build`

To watch the changes of typescript files in server directory use `js static $ npm run tsc-watch`

To start the server and watch changes made to javascript files run `js static $ npm run watch` You normally want to run those two commands above at the same time. One is to watch and compile typescript files and the other is to watch if there are any changes to compiled javascript files, rerun the server.

E.5. How to generate documentation

To watch for changes and dynamically build interactive react documentation `js static $ npm run styleguide`

To build a static version of the interactive react documentation `js static $ npm run styleguide:build`

To build a markdown version of the react documentation `js static $ npm run react-docs`

To build a markdown version of the JavaScript documentation `js static $ npm run js-docs`

To build a markdown version of the TypeScript documentation `js static $ npm run ts-docs`

To build markdown versions of all documentation (Note: they can be found in the /documentation/ directory) `js static $ npm run docs`

F. MARAS Documentation

This repository (https://bitbucket.org/divamqp/diva-maras) contains the code for the Multi-Drug Adverse Reaction Analytics Strategy (MARAS) system.

The MARAS directory contains the java code for the MARAS system, including preprocessing, drug and reaction parsing, data mining, and output for visualization (see https://bitbucket.org/divamqp/diva-node-web).

The diva-docs repository contains the markdown files for our report.

F.1. How to run

The MARAS system is used with the DIVA platform. The analysis runs when a FAERS data file is uploaded, and the visualization is updated accordingly.

The MARAS system can be manually run using the java code, running App.java. Command line arguments are as follows:

-f: FAERS .zip file to analyze

-d: Path to the Drug map file to be used (should usually point to the test2.json file contained in the data directory)

-r: Path to the Standardized Known Rules file to be used (should usually point to the knownRules_standardized.csv file contained in the data directory)

-m: Path to the MetaMap installation directory

-s: Minimum support for rules to be considered

-c: Minimum confidence for rules to be considered

-o: Output Path

-p: Path to the reports file to add on to (for the Visualization)

-t: Path to the status file (for the Visualization)

MetaMap processes should be running separately when the system is started.

To best emulate the process used by the DIVA system, use the maras.sh wrapper script contained in the DIVA repository, which handles launching MetaMap automatically.

F.2. Output

The MARAS system outputs multiple files.

rules.csv: This is the rule data that is used by DIVA to visualize the results. In the DIVA-MARAS workflow it is output to the storage directory, which is where the DIVA system looks for this data.

closed_results.txt: This file gives a detailed view of all of the rules mined, sorted from highest score to the lowest. Each rule displays the drugs and the associated ADR, along with the contrast score and the rule's support and confidence. In addition, it shows the support and confidence of each contained drug mapped to the ADR, to represent the Context Associated Cluster (CAC).

_rules_drug_adr_matching.txt: This shows the same rule data in rules.csv in a different format. _rules_drug_adr_matching_long_names.txt shows the same data, but with the original drug names rather than the matched ones.

F.3. Data

The data directory contains much of the information used by the system. Reports.txt is a file containing all report information, used by the DIVA platform in order to maintain relevent report data and have it available for FDA evaluators. The ZipFiles directory contains two sample FAERS data sets: the 2013Q1 and 2017Q2 data. The raw directory is where the underlying text files from the FAERS data sets are extracted; if there are duplicates here, the system will end the process, recognizing that the data has already been analyzed.

For each data set, the report information is converted to a more brief and compact format, and written to a text file with a name corresponding to the data set; for example the 2013 Q1 report data will be exported to 2013Q1Reports.txt. This is so when additional data is added, there is no need to repeat FAERS data extraction and analysis.

If you would like to run analysis on a set of data by itself, simply remove the associated Reports files from the raw directory, and remove the truncated report file (e.g. 2013Q1Reports.txt) for any data you do not want included. Otherwise, the included data set will be used in combination with the previously analyzed data. MARAS (Java) Documentation ==========

F.4. `public class App`

Main Application class for the MARAS system. This class handles arguments, input data parsing, and output formatting.

- **Author:** Brian

F.4.1. `public static void main(String[] args)`

Main function to handle parameters

- **Parameters:** `args` — The arguments passed

F.4.2. `public void run()`

Begin MARAS execution

F.4.3. `public static void handleException(Exception e)`

Static function to handle an exception and gracefully exit the execution. This function updates the status file to properly include the error message. It also prints out the error details to Standard Output to ensure it is properly written to the log.txt file. This function should be called any time an Exception is caught that should kill the program.

- **Parameters:** `e` — The Exception thrown

F.4.4. `private File parseFaersFiles(List<File> files, File outputDir) throws IOException`

Parse a list of FAERS data files, returning an output file organizing each report

- **Parameters:** `files` — The FAERS data files (DRUG, REAC, and DEMO)
- **Returns:** The output file
- **Exceptions:** `IOException` —

F.5. `public class Dataset`

The class defining the Dataset structure. A Dataset is a collection of {Report}s where each report is generated from a single Adverse Reaction FARAS report.

- **Author:** Andrew Schade

F.5.1. public Dataset(List<File> reportFiles, int maxTransactionLength, double minSupport, double minConfidence, File reportOutput) throws FileNotFoundException, UnsupportedEncodingException

Reads in a dataset from a file.

- **Parameters:**
- drugFiles —
- path — The location of the file containing the dataset data.
- **Exceptions:**
- FileNotFoundException —
- UnsupportedEncodingException —

F.5.2. public InteractionSets getInteractions()

Generates itemsets after checking if the itemsets are already generated, if they are it will return the generated itemests.

- **Returns:** The {InteractionSets} mined with {FPGrowth}.

F.5.3. public RuleSets getAssociationRules()

Generates rulesets after checking to see if the rulesets are already generated. If they are, it will return the generated rulesets. Also will generate itemsets if those are not already generated.

- **Returns:** The {RuleSets} mined with {Agrawal}.

F.5.4. public int getDatasetSize()

Gets the number of reports in the database.

- **Returns:** The size of the database defined by the number of reports.

F.6. public class Drug extends Item

Class representing a Drug This class maintains a static list of all drugs used in the system

- **Author:** Andrew Schade

F.6.1. `protected Drug(int ID, String shortName, String fullName)`

Private Constructor, just initializes drug list.

F.6.2. `public static void readDrugs(String path) throws JsonIOException, JsonSyntaxException, FileNotFoundException`

Read drugs from the drug map file to initialize drugs with their IDs

- **Parameters:** `path` — The path to the drug file
- **Exceptions:**
- `JsonIOException` —
- `JsonSyntaxException` —
- `FileNotFoundException` —

F.6.3. `private static String process(String name)`

Process a drug name to remove any key phrases or words contained in the wordsToRemove list

- **Parameters:** `name` — The drug name
- **Returns:** The processed drug name

F.6.4. `public static Drug match(String name)`

Matches a drug name to a pre-existing drug object

- **Parameters:** `name` — The name of the drug
- **Returns:** The drug object

F.7. `public class Group implements Comparable<Group>`

A Group contains a complete ADR rule and its singleton component associations.

F.7.1. `public Group(Rule comb)`

Creates a group from a combined rule. Populates the bins for singletons, but does not find them. Does not calculate the group score.

- **Parameters:** `comb` — The rule to use as a basis for the group.

F.7.2. `public void setKLDs(double[] klds)`

Set the kl divergences to a new array.

- **Parameters:** `klds` — A list of KL divergence values.

F.7.3. `public double[] getKLDs()`

- **Returns:** the array of kl divergences.

F.7.4. `public void calcScore()`

The algorithm for calculating the score. Updates the score in the group but not the composite rule.

score = ((overall_confidence - average_confidence) * (1 - (conf_std_dev / avg))) / number_of_comb

F.7.5. `public double getScore()`

- **Returns:** The group score.

F.7.6. `public void setScore(double score)`

Set the score of a group. Used for testing to ensure that the groups generated match those from another file.

F.7.7. `public Rule getCompositeRule()`

- **Returns:** The composite rule for the group.

F.7.8. `public ArrayList<HashSet<Rule>> getSingletonRules()`

- **Returns:** An arraylist of the Hashset for singleton rules.

F.7.9. `@Override  public int compareTo(Group g)`

Comparator for descending sorting of groups. This comparator behaves opposite to convention.

F.7.10. `public void addSingle(Rule rule)`

Adds a singleton to the appropriate bin.

- **Parameters:** `rule` — The singleton rule to add.

F.8. `public class Groupset extends ArrayList<Group>`

This class contains a set of groups developed from a closed ruleset. It provides an interface for creating output files in formats useful for DIVA visualization and human consumption.

- **Author:** Andrew Schade

F.8.1. `public String getResultString()`

Get a string representing the results of analysis

- **Returns:** The string

F.8.2. `public String getResultString(int top)`

Get a string representing the results of analysis

- **Parameters:** `top` — Max number of results to consider
- **Returns:** The String

F.8.3. `public String getVizString()`

- **Returns:** The visualization string

F.8.4. `public String getVizString(int top)`

Get the string used by the visualization to represent the data

- **Parameters:** `top` — The max number of results
- **Returns:** The visualization string

F.8.5. `public String getVizLabelString()`

Visualization data file generation

- **Returns:** The string

F.8.6. `public String getVizLabelString(int top)`

Visualization Data file Generation.

- **Parameters:** `top` — The max number of results to consider
- **Returns:** The string

F.8.7. `List<String> matchingReportIDs = new ArrayList<String>()`

To find a list of supporting reports, we check to find the list of Report IDs that are similar across all of the Items

F.8.8. `public void outputDIVARules(BufferedWriter outfile) throws IOException`

Write the DIVA rules

- **Parameters:** `outfile` — The output file to write to
- **Exceptions:** `IOException` —

F.8.9. `private String getDIVARuleString()`

Get the DIVA rule string

- **Returns:** The rule string

F.8.10. `private String getDIVARuleString(int top)`

Get the DIVA rule string

- **Parameters:** `top` — The max number of results to consider
- **Returns:** The rule string

F.8.11. `private String getHeaderRow()`

Get the header string for the output

- **Returns:** The header string

F.8.12. `public void outputGroups(BufferedWriter outfile) throws IOException`

Write the results to file

- **Parameters:** `outfile` — The file to write to
- **Exceptions:** `IOException` —

F.8.13. `public void outputViz(BufferedWriter labels, BufferedWriter info) throws IOException`

Write the visualization strings to file

- **Parameters:**
- `labels` — The label file to write to
- `info` — The info file to write to
- **Exceptions:** `IOException` —

F.8.14. `public void outputViz(BufferedWriter outfile) throws IOException`

Write the visualization string to file

- **Parameters:** `outfile` — The file to write to
- **Exceptions:** `IOException` —

F.9. `public class Interaction extends Itemset`

This class represents a set of items and a support count. This implementation is based on the SPMF Itemset data type by Philippe Fournier-Viger.

- **Author:** Brian McCarthy

F.9.1. `public void addItem(Item item)`

Add an item to the list

- **Parameters:** `item` — The item to add

F.9.2. `public Interaction cloneMinusOne(Item item)`

Get an interaction with the list of items excluding the given one

- **Parameters:** `item` — The item to exclude
- **Returns:** The cloned interaction

F.9.3. `public Interaction cloneMinusInteraction(Interaction interaction)`

Get an Interaction with all of the items NOT contained in the given interaction

- **Parameters:** `interaction` — The interaction to exclude items from
- **Returns:** The cloned interaction

F.10. `public class InteractionSets extends Itemsets`

This class represents a set of Interactions, where an Interaction is a set of items with an associated support count. Interactions are ordered by size. For example, interactionSet 1 is a set of interactions of size 1 (contains 1 item).

Based on SPMF Itemsets data type by Philippe Fournier-Viger

- **Author:**
- Brian McCarthy
- Andrew Schade

F.10.1. `public InteractionSets getClosures()`

This method finds all the interactions within this collection where for all k greater than the level of a given interaction, the interaction is a member of the closure if and only if the interaction is not a subset of any interaction with a level of k.

- **Returns:** The K-interactionsets with only closed interaction contained inside.

F.11. `public class Item implements Comparable<Item>`

A class defining the Item structure. It also contains a map for associating Items with their unique IDs.

- **Author:** Andrew Schade

F.11.1. `public static void restartMetaMap()`

Method to relaunch the MetaMap API

F.11.2. `public static Item fromInt(int i)`

Finds the Item defined by the specified unique ID.

- **Parameters:** `i` — The unique ID query to find an Item.
- **Returns:** The item matching the unique ID query. If the ID is not matched a NoSuchElementException will be thrown.

F.11.3. `public static Item fromIntKnown(int i)`

Finds the Item defined by the specified unique ID. Doesn't have a lock because no new data is being added, only look-ups are happening.

- **Parameters:** `i` — The unique ID query to find an Item.
- **Returns:** The item matching the unique ID query. If the ID is not matched a NoSuchElementException will be thrown.

F.11.4. `public static int itemID(String name, boolean isDrug, boolean standardized)`

Gets the item ID for a specified string.

- **Parameters:** `name` — the string name of the item. Will be matched ignoring case.
- **Returns:** The integer ID matched for the string. If multiple items have the same name, the lowest ID match will be returned. A NoSuchElementException will be thrown if there is no matched string.

F.11.5. `public static int itemIDKnown(String name, boolean isDrug, boolean standardized)`

Gets the item ID for a specified string. Doesn't have a lock because no new data is being added, only look-ups are happening.

- **Parameters:** `name` — the string name of the item. Will be matched ignoring case.

- **Returns:** The integer ID matched for the string. If multiple items have the same name, the lowest ID match will be returned. A NoSuchElementException will be thrown if there is no matched string.

F.11.6. `public static Item makeItem(String name, boolean isDrug, boolean standardized)`

This is the public constructor for creating an Item. It loads an item in from a name. This can be updated in the future to ensure ID consistency. If a name already maps to an item, then it doesn't create a new one.

- **Parameters:** `name` — The name of the drug or reaction, etc.
- **Returns:** A new item created, which is added to the data structure tracking all items.

F.11.7. `public static Item checkADRName(String name, boolean createNew)`

Checks to see if an adr corresponds to an alternate name for a known adr. If so, returns the preferred name for the adr.

- **Parameters:**
- `name` — The unformatted name of an ADR
- `createNew` — Boolean flag indicating whether a new ADR item should be created if it does not already exist. If false, a NoSuchElementException is thrown.
- **Returns:** Preferred name for adr

F.11.8. `public static int getNumDrugs()`

Gets the number of distinct items that are drugs.

- **Returns:** The number of distinct items that are drugs.

F.11.9. `public static int getNumReactions()`

Gets the number of distinct items that are adr's.

- **Returns:** The number of distinct items that are adr's.

F.11.10. `@Override public String toString()`

Gets the standardized version of the item's name. I.e. all spaces are removed and all letters are lower case.

return Standardized version of the item's name.

F.11.11. `protected Item(String name, String longName, int ID, boolean isDrug)`

The private constructor for Item. This is pretty straightforward. It contains the name of the item and its unique ID.

- **Parameters:**
- `name` — The name of the drug or reaction.
- `ID` — The unique ID referring to the Item.

F.11.12. `public boolean isDrug()`

A getter object to determine if this item is a drug or a reaction.

- **Returns:** true if drug, false if reaction

F.11.13. `public int getiD()`

A getter object to receive the unique ID of the Item.

- **Returns:** the Item's unique ID.

F.11.14. `public String getShortName()`

A getter object to receive the short name of the Item

- **Returns:** the Item's name.

F.11.15. `public String getLongName()`

A getter object to receive the long name of the Item

- **Returns:** the Item's name.

F.11.16. `@Override public int compareTo(Item arg0)`

Used to determine if two items are the same by comparing their IDs.

- **Parameters:** `arg0` — The item being compared with this item.
- **Returns:** Integer indicating that this item's ID comes before (-1), after (1), or is the same as (0) the other item's ID.

F.12. `public class MarasStatus`

Function representing the status of the system This class is used to simplify the JSON management

- **Author:** Brian

F.13. `public class Report implements Comparable<Report>, Iterable<Item>`

A report maps to a single entry in the FARAS data recovered for use.

- **Author:** Andrew Schade

F.13.1. `@Override public int compareTo(Report other)`

This function compares two reports to each other, allowing sorting by ascending transaction length. In cases where the transaction length is the same, the ordering will use the ordering determined by the drugs and reactions in the reports, starting with the first.

F.14. `public class Rule extends AssocRule implements Comparable<Rule>`

This class represent a MARAS association rule. A rule can be used to find a group (a rule and all singleton association rules that can be derived from that rule).

This class is an extension of the AssocRule class from the SPMF library put together by Philippe Fournier-Viger.

- **See also:**
- AssocRule
- Item
- Group

- RuleSets
- **Author:** Brian Zylich

F.14.1. `private static Map<Rule, Boolean> knownRules = new HashMap<Rule, Boolean>()`

Used to hold all known drug-drug interactions

F.14.2. `public static void parseStandardizedKnownRulesThreading(String fileName)`

Reads in a set of known rules that have already been converted to standardized forms. This function uses threading to accomplish this goal.

- **Parameters:** `fileName` — The path of the file containing the standardized known rules.

F.14.3. `public static void standardizeAllKnownRulesThreading(String fileName)`

Reads in a set of known rules that have not already been converted to standardized forms. Then, the standardized known rules are written to a file. This function uses threading to accomplish this goal. This function should only be run once or whenever the known rules file changes and a new standardized known rules file must be generated.

- **Parameters:** `fileName` — The path of the file containing the known rules.

F.14.4. `private static void addKnownRule(List<Item> antecedent, List<Item> consequent)`

Adds a new rule, created from the given antecedent and consequent, to the map of known rules. This function is protected by a lock to allow for multithreading.

- **Parameters:**
- `antecedent` — A list of items corresponding to the drugs involved in the rule.
- `consequent` — A list of items corresponding to the reactions involved in the rule.

F.14.5. `private List<Integer> reportIds`

Used to track which FAERS reports support this rule

F.14.6. `private final List<Item> antecedent`

List of Item objects forming the antecedent

F.14.7. `private final List<Item> consequent`

List of Item objects forming the consequent

F.14.8. `private boolean known = false`

Boolean indicating if the rule corresponds to a known drug-drug interaction

F.14.9. `public Rule(List<Item> itemset1, List<Item> itemset2, int supportAntecedent, int transactionCount, double confidence, double lift)`

Constructor

- **Parameters:**
- `itemset1` — the antecedent of the rule (an itemset)
- `itemset2` — the consequent of the rule (an itemset)
- `supportAntecedent` — the coverage of the rule (support of the antecedent)
- `transactionCount` — the absolute support of the rule (integer)
- `confidence` — the confidence of the rule
- `lift` — the lift of the rule

F.14.10. `private Rule(List<Item> itemset1, List<Item> itemset2)`

Constructor - only to be used for known DDRs (don't care about their support, confidence, etc)

- **Parameters:**
- `itemset1` — the antecedent of the rule (an itemset)
- `itemset2` — the consequent of the rule (an itemset)

F.14.11. `private static int[] getItemIds(List<Item> itemList)`

Helper function used to turn a list of Items into an array of integers representing those items.

- **Returns:** the array of integer IDs.

F.14.12. `public List<Item> getAnte()`

Returns the list of item objects representing the antecedent of the rule.

- **Returns:** the list of item objects.

F.14.13. `public List<Item> getCons()`

Returns the list of item objects representing the consequent of the rule.

- **Returns:** the list of item objects.

F.14.14. `public Group getGroup(RuleSets rules)`

Creates a Group from this Rule that includes this Rule and all subrules that can be derived from this Rule.

- **Returns:** the Group created from this Rule.

F.14.15. `public Interaction getInteraction()`

Used to find the interaction corresponding to this rule.

- **Returns:** InteractionSet containing all items from the antecedent and the consequent of the rule.

F.14.16. `public List<Integer> getReports()`

Used to find which FAERS reports support this association rule.

- **Returns:** list of report ids.

F.14.17. `public void checkKnown()`

Used to check whether a rule is known or unknown, modifying the 'known' variable within the Rule object. This function should be run after loading all known rules.

F.14.18. `public boolean isKnown()`

Used to determine if this rule corresponds to a known drug-drug interaction

- **Returns:** boolean indicating if rule is known

F.14.19. `private Rule getSimpleRule()`

Used to compare a newly generated rule against known rules by simplifying the rule for hashing.

- **Returns:** Simplified version of the rule containing only the antecedent and consequent

F.14.20. `public boolean sameRule(Rule other)`

Check whether two rules have the same items in their antecedent and consequent

- **Parameters:** `other` — The other rule to compare to
- **Returns:** boolean indicating whether the antecedents and consequents match

F.14.21. `@Override  public boolean equals(Object other)`

Check whether two rules have the same items in their antecedent and consequent, as well as the same support and confidence values.

- **Parameters:** `other` — The other rule to compare to
- **Returns:** boolean indicating whether the rules are the same

F.14.22. `@Override public String toString()`

Get a string representation of the rule, including the rule's antecedent, consequent, support, and confidence.

- **Returns:** string representation of the rule

F.14.23. `@Override public int compareTo(Rule r)`

Comparator used to sort rules in groups for hashing and to facilitate testing.

F.15. `public class RuleSets extends AssocRules`

This class represent a set of MARAS association rules. RuleSets can be used to find closures (filters rules to ensure that the itemset crated by the rules is a closure).

This class is an extension of the AssocRules class from the SPMF library put together by Philippe Fournier-Viger.

- **See also:**

- AssocRules
- InteractionSets
- Rule
- **Author:** Brian Zylich

F.15.1. `public RuleSets(String name)`

Constructor

- **Parameters:** `name` — a name for this list of association rules (string)

F.15.2. `public RuleSets(String name, AssocRules rules)`

Constructor

- **Parameters:**
- `name` — a name for this list of association rules (string)
- `rules` — rules generated by the association rule mining algorithm

F.15.3. `public RuleSets findClosures(InteractionSets closures)`

Filters the rules to ensure that the itemset created by the rules is a closure.

- **Parameters:** `closures` — The InteractionSets object containing all closed InteractionSet's.
- **Returns:** RuleSets object containing all of the closed rules.

F.15.4. `public RuleSets filterRules()`

Filters the rules to ensure that the antecedent only contains drugs and the consequent only contains reactions.

- **Returns:** RuleSets object containing all of the filtered rules.

F.15.5. `public RuleSets filterNoSingletonRules()`

Filters rules to remove rules with <2 drugs

- **Returns:** RuleSets object without any rules that have less than 2 drugs

F.15.6. `public RuleSets filterNoComplexRules()`

Filters rules to remove rules with >2 drugs or >1 ADR

- **Returns:** RuleSets object without any rules that have more than 2 drugs or more than 1 adr

F.16. `public class Agrawal extends AlgoAgrawalFaster94`

This class extends the AlgoAgrawalFaster94 class from the SPMF library put together by Philippe Fournier-Viger. The extension of this association rule mining algorithm is used to run the algorithm using InteractionSets and producing RuleSets.

- **See also:**
- AlgoAgrawalFaster94
- InteractionSets
- RuleSets
- **Author:** Brian Zylich

F.16.1. `public Agrawal()`

Constructor

F.16.2. `public RuleSets runAlgorithm(InteractionSets interactions, int databaseSize, double minconf)`

Run the algorithm

- **Parameters:**
- `interactions` — a set of frequent itemsets
- `databaseSize` — the number of transactions in the database
- `minconf` — the minconf threshold
- **Returns:** the set of association rules