MY LEARNING JOURNEY

SALMAN ALMUZAINI

Abstract. This document serves as a comprehensive record of my exploration into utilizing Rust for data science purposes. It details the chronological progression of my experience, starting from the initial setup with Rust's package manager, Cargo, to overcoming the various challenges encountered along the way. The narrative encapsulates my endeavours in API integration, error handling, and the utilization of Docker for environment management, highlighting the intricacies of cross-compiling and static linking within this context. Reflections on the project scope, adaptability in learning, and strategic decision-making are also discussed. The journey unfolds across multiple days, each presenting unique obstacles and learning opportunities —from Rust's compilation nuances to the complexities of Apache Arrow for data management. This account not only charts technical milestones but also underscores the iterative and adaptive nature of programming and data science within the Rust ecosystem.

0.1. **Day 1: Rust, APIs, and Containerization Challenges.**

Embarking on a journey to master Rust for data science, I set my sights on leveraging APIs for dynamic data retrieval. The KAPSARC (King Abdullah Petroleum Studies and Research Center) API promised a wealth of energy data, but tapping into this resource proved to be a complex task.

0.1.1. *Setting Up with Cargo.toml.*

The first order of business was configuring the Cargo.toml to declare the dependencies necessary for making HTTP requests (reqwest) and handling JSON data (serde). After overcoming an initial hiccup where reqwest wasn't recognized, a corrected Cargo.toml looked like this:

```
[dependencies]
reqwest = "0.11"
serde = "1.0"
serde_json = "1.0"
```

This setup was my gateway to the world of Rust dependencies.

0.1.2. *The Build: Rust's Compilation Dance.*

Executing cargo build set off a cascade of downloads and compilations, a testament to the rich ecosystem Rust offers. Watching libraries like cfg-if and percent-encoding get ready for action was my first taste of victory.

0.1.3. *The Runtime: A Tryst with Errors.*

However, my celebration was short-lived as cargo run threw a stark runtime error — a missing file. It underscored the importance of crafting precise error handling in Rust. Adjusting the error handling logic, I braced myself for the next attempt.

But the hurdles continued. The API's JSON response was missing the total_-count field, expected by my parsing logic. Rust's Option type came to the rescue, allowing me to handle the unpredictability of real-world data:

```
#[derive(Deserialize)]
struct ApiResponse {
    total_count: Option<i32>, // Now optional, no missing field error
    // Other fields as required
}
```

This adjustment was a key learning moment in data flexibility and error resilience.

0.1.4. *Docker: The Container Conundrum.*

As the project progressed, I decided to encapsulate the development environment using Docker, with Alpine Linux as the base image for its minimalistic footprint. This decision led to a confrontation with a new beast: building Rust applications for musl libc instead of the default glibc. This choice, aimed at creating a more secure and portable executable, introduced me to the challenges of cross-compiling Rust applications.

The openssl-sys crate, necessary for HTTPS requests, became the center of my struggle as it clashed with the musl-based environment. The errors were obscure, leading to a deep dive into Alpine's package manager and Rust's compilation flags. The learning curve was steep, as I navigated through the intricacies of static linking and system dependencies.

0.1.5. *Project Scope and Reflections.*

Amidst the technical battles, the scope of my project began to shift. I had to reassess my goals and adapt my approach to align with my growing understanding of Rust's capabilities and limitations. The initial intention to use Jupyter notebooks for an interactive Rust experience was shelved in favor of focusing on the API integration aspect.

Day 1 was not just about code; it was about adaptability, problem-solving, and continuous learning. It reinforced the iterative nature of programming, where each error leads to a deeper insight, and every decision shapes the trajectory of the project.

In the rearview mirror, the day was a confluence of technical challenges, strategic pivots, and the inevitable realization that learning Rust was as much about understanding its ecosystem as it was about the language constructs themselves. It was a day of laying the groundwork for a journey that was only just beginning.

0.2. **Day 2: Dockerization and Execution Woes.**

With the foundations set on Day 1, Day 2′s mission was to containerize the Rust environment using Docker to ensure a consistent development and execution environment. This was done with the aim of creating a lightweight and secure setup that could potentially be deployed anywhere with ease.

0.2.1. *Docker Build: A Smooth Sail.*

The day began with a sense of optimism as I initiated the Docker build process. The Dockerfile was designed to use Alpine Linux as the base image, keeping the overall footprint small. The build sequence was crafted carefully to ensure that all Rust tooling and dependencies were in place:

The curl command was used to test the API connectivity within the container. The jq tool was installed to allow for command-line JSON processing. A new Rust project RustDataVoyager was created within the container. The project source code was copied into the container. A final cargo build command was invoked to compile the Rust application. The Docker image build log showed no errors, and the tag dataquest-rust-docker:latest was successfully applied to the image. It was a moment of accomplishment as the image was now ready to be instantiated into a container.

0.2.2. *Execution Hurdles: The Binary Conundrum.*

However, my excitement was tempered when I attempted to run the newly created Rust application inside the Docker container. The command cargo run failed, with the error message indicating that the target/debug/rust_data_voyager binary could not be executed due to it not being found. This was perplexing as the binary should have been created during the build process.

I was dropped into an Alpine Linux shell prompt inside the container to further investigate the issue. Commands like uname -a confirmed the Linux environment, while a listing of the RustDataVoyager directory showed all the expected files in place, yet the binary remained elusive.

0.2.3. *Troubleshooting: A Deep Dive.*

The troubleshooting process was meticulous:

I reviewed the build output during the Docker image build to ensure there were no hidden errors. I verified that the Dockerfile and build process were supposed to compile the Rust application and generate the binary. I checked the target/ debug directory within the source code to look for the binary. I made sure that the Dockerfile set the working directory correctly to /usr/src/RustDataVoyager. Each step was taken with a careful approach to uncover the root cause of the missing binary. The process involved a lot of back-and-forth, examining build logs, and verifying directory paths.

0.2.4. *Reflections and Learnings.*

Day 2 brought with it a stark reminder of the complexities of containerization and cross-environment development. While Docker promised a uniform environment, it also introduced new variables that required attention to detail and a deep understanding of both the Rust ecosystem and Docker's mechanisms.

Despite the setbacks, the day was rich with learning opportunities. It underscored the importance of verifying every step of the development pipeline, from writing code to deploying executables. It was a day that tested patience and perseverance but ultimately contributed to a growing repository of knowledge and skills.

0.3. **Day 3: Rust Error Handling and Data Management.**

On Day 3, I confronted a critical juncture in my Rust learning journey. The initial idea of leveraging a live API for data science tasks was met with unexpected challenges. The API proved to be unreliable, or perhaps it was my inexperience with its intricacies that led to inconsistent results. In response, I pivoted my approach:

- Reliability: I opted for downloaded CSV files, which offered a more stable and reliable data source than the unpredictable API.
- Simplicity: This shift allowed me to focus my efforts on data processing with Rust, without the added complexity of API management.
- Control: Static CSV files provided a controlled environment for a more straightforward debugging and learning process.

Acknowledging the steep learning curve with the API, I redirected my project to work with CSV files, setting the stage for a more focused and fruitful exploration of data science with Rust.

0.3.1. *The Error Conundrum: Rust's Type System at Play.*

The day's work began with an error handling issue that Rust's compiler brought to light. My attempt to create a file and write to it within a function that returns a Result<(), reqwest::Error> resulted in compilation errors. The compiler was unsparing in its feedback:

```
let mut file = File::create(&file_path)?;
file.write_all(&bytes)?;
The ? operator, which I had used to propagate errors up the call stack,
was designed to work with compatible error types. However, the errors
arising from file operations (std::io::Error) were not automatically
convertible to the expected reqwest::Error. This was Rust's type system
enforcing strict error type consistency.
```

I learned that I had to map the error to the appropriate type or change the function's return type to accommodate multiple error types using an enum or a boxed error type. This was a deep dive into Rust's powerful, yet sometimes complex, error handling model.

0.3.2. *Data Export Trials: CSV Format Handling.*

The day also saw me exploring data exportation. I had been interacting with the KAPSARC API, trying to export datasets. After some trial and error, I decided to attempt exporting a dataset in CSV format. Using the curl command, I fetched the data and saved it to "main.csv". The command looked like this:

```
curl -X 'GET' 'https://datasource.kapsarc.org/api/explore/v2.1/catalog/
datasets/the-renewable-energy-policy-paradox/exports/csv' -H 'accept:
application/json; charset=utf-8' -o main.csv
```

The CSV file, once opened, presented a new challenge — it contained metadata headers that needed parsing to access the actual dataset content:

```
recordid;_record_id;record_timestamp;_record_timestamp;...
I learned that dealing with real-world data often involves an additional
layer of processing to separate useful information from metadata. This
was an important lesson in data cleaning and preparation, essential
skills for any data scientist.
```

0.3.3. *User Interaction and Support.*

As part of the learning experience, I also engaged in discussions about how to handle the CSV data. The headers in the CSV seemed to pertain to metadata rather than the data itself. I advised on opening the file in a text editor or spreadsheet program to inspect the content and find the actual dataset data.

This interaction was a reminder of the importance of good communication skills in programming — being able to clearly explain and discuss issues is as crucial as solving them.

0.3.4. *Reflections on Progress.*

The third day brought with it a recognition of the multifaceted nature of programming. It was not just about writing code; it was also about understanding the ecosystem in which the code operates and the data it manipulates. Error handling, data processing, and user support — each aspect was a piece of the larger puzzle.

The day concluded with a sense of accomplishment and a deeper appreciation for Rust's capability to enforce robustness through its strict type system. The journey through Rust was proving to be as much about learning the language as it was about understanding the principles of software development and data science.

0.4. **Day 4: Transitioning Environments and Refining Data Handling.**

The fourth day of my data science journey with Rust was one of significant transitions and deep technical exploration. Embracing substantial changes to my development setup, I tackled the intricacies of data manipulation and began to reshape my project's structure for enhanced maintainability and scalability.

0.4.1. *Workflow Shift: From Docker to Local Development.*

A pivotal shift occurred in my workflow as I moved away from Docker and Alpine Linux containers. I opted to run Rust locally on my Mac, which provided immediate benefits:

Local Environment: The integrated development process and immediate feedback loop improved my efficiency and speed. Simplicity: I removed the overhead of Docker management, streamlining my focus onto Rust itself. Performance: By avoiding the potential limitations of containerized setups, my Mac's resources were better utilized. This transition represented a growing understanding of my development tools and a commitment to optimizing the learning environment.

0.4.2. *Project Restructuring: Modularization for Clarity.*

To address the increasing complexity of my project, I reorganized my codebase into several distinct modules:

concatenation.rs: Handled the merging of disparate data sources. arrow_converter.rs: Encapsulated the logic for converting CSV data into the Apache Arrow format. data_analysis.rs: Contained functionalities for data analysis and insights extraction. data_validation.rs: Focused on ensuring data integrity and accuracy. The introduction of a lib.rs file allowed these modules to be orchestrated cohesively, with main.rs now acting as the entry point that called upon the library for specific functionalities.

0.4.3. *CSV Parsing: Challenges and the Apache Arrow Paradigm.*

A major technical hurdle was a schema mismatch error in CSV parsing:

```
Error: InvalidArgumentError("number of columns(1) must match number of
fields(47) in schema")
```

This error led to the refinement of the create_arrow_schema function, ensuring an accurate reflection of the CSV structure. It also prompted a strategic adoption of the Apache Arrow format, chosen for its scalability and its suitability for emulating real-world data scenarios.

Apache Arrow's efficient columnar storage model was a key factor in my decision to mirror production data workflows, reaffirming my dedication to applying the right tools for realistic data science challenges.

0.4.4. *Data Science with Rust: A Comparative Perspective.*

I dedicated time to comparing Rust's data handling capabilities with Python's, focusing on tasks such as CSV file reading, missing value handling, and data type manipulation. Rust offered explicit control, while Python provided ease of use—a contrast that became increasingly apparent.

0.4.5. *Reflecting on Language Efficacy and Future Directions.*

Day 4 was an enlightening chapter in my Rust journey, as I balanced high performance with developer productivity. The day's experiences solidified my understanding of Rust's potential in data handling and prepared me for the upcoming challenges of my project.

0.5. **Day 5: Navigating Through Compilation Errors and Embracing Apache Arrow.**

On Day 5, the complexities of Rust's type system and the Apache Arrow library were at the forefront of my journey.

0.5.1. *Compilation Errors: A Steep Learning Curve.*

My main task for the day was to validate and convert CSV data to the Apache Arrow format. This required a deep dive into the Arrow IPC format and handling of record batches. However, I encountered several compilation errors that stumped me:

Type Mismatch Error: The compiler flagged an InvalidArgumentError because the number of columns in my CSV didn't match the number of fields in the Arrow schema. The error message was:

```
Error: InvalidArgumentError("number of columns(1) must match number of
fields(47) in schema")
```

To tackle this, I had to revise my code to ensure that the Arrow schema was dynamically aligned with the CSV structure.

Incorrect Method Calls: I mistakenly called a method that didn't exist on the FileReader struct in the Arrow crate. The error message was:

```
error[E0599]: no method named `open_batch` found for struct `FileReader`
in the current scope
```

This led me to carefully review the Arrow crate's documentation to find the correct method, get_record_batch, and to understand the proper way to iterate over batches.

Incorrect Error Propagation: I used the ? operator in a context where it wasn't applicable, which prompted the following error message:

```
error[E0277]: the `?` operator can only be applied to values that
implement `Try
```

Resolving this required a better understanding of Rust's error handling patterns, particularly when dealing with iterators and non-Result types.

0.5.2. *Embracing Apache Arrow for Scalability.*

Despite these challenges, I remained focused on my goal to convert CSV data to the Arrow format. My motivation was clear: Apache Arrow offers a scalable way to handle large datasets, essential for real-world data science tasks. The columnar memory format is optimized for modern CPUs and large-scale operations, making it a suitable choice for data-intensive applications.

By the end of the day, I had made significant progress. I updated my data_validation.rs file with the necessary changes to address the errors and warnings:

rust This not only fixed the compilation issues but also reinforced my understanding of Rust's robust type system and the importance of precise coding.

0.5.3. *Reflecting on the Day's Progress.*

Day 5 was challenging, yet it was a testament to the iterative nature of software development. Each error message, while initially daunting, provided a valuable learning opportunity. It also highlighted the importance of Apache Arrow in the data science workflow, particularly for scalability and performance.

As I concluded the day, I felt more confident in my ability to navigate Rust's complexities and in the decision to use Apache Arrow for processing large-scale data.

0.6. **Day 6: Refinement and Error Resolution in CSV to Arrow Conversion.**

Day 6 of my Rust for data science journey was dedicated to refining the CSV to Apache Arrow conversion process, with a focus on resolving compilation errors and improving data validation.

By Day 6, I was fully immersed in refining the CSV to Apache Arrow conversion process. With the project scope change on Day 3 already established, I could dedicate my efforts to overcoming the technical complexities of Rust and the Arrow library.

- Error Handling: I tackled type mismatches and data parsing errors head-on, each one offering new insights into Rust's strict type safety and error handling.
- Data Conversion: The conversion process from CSV to Apache Arrow format became smoother as I applied the lessons learned from previous days,

appreciating the efficiency and scalability of the Arrow format for big data tasks.
- Technical Insight: Day 6 was marked by a series of breakthroughs in understanding and utilizing Rust's powerful features for data processing.

Reflecting on the decision made on Day 3, the benefits of working with CSV files were clear. It provided a more predictable and manageable environment for mastering Rust's data science tools, setting a strong foundation for future developments.

0.6.1. *Type Handling and Schema Validation.*

I encountered a type mismatch error that prevented the successful compilation of my code. The issue was with the handling of an Arc object, a reference-counted pointer used in Rust to allow multiple ownership of data, which is especially useful for complex data structures like Apache Arrow schemas. The error message was clear:

```
Error: InvalidArgumentError("number of columns(1) must match number of
fields(47) in schema")
```

This pointed to a mismatch between the expected schema and the CSV file's actual structure. To resolve this, I updated the validate_csv_input and convert_csv_to_arrow functions to ensure they handled the Arc correctly, providing a consistent and accurate schema for the data conversion process.

0.6.2. *Debugging Data Parsing Issues.*

Another challenge was a ParseFloatError, indicating that the CSV data contained fields that could not be parsed into floating-point numbers as expected by the schema. This error highlighted the importance of robust data validation and error handling in data processing pipelines:

```
Error: ParseFloatError { kind: Invalid }
```

To address this, I had to carefully inspect the CSV file to ensure that all float fields were in the correct format and did not contain any invalid characters or formats. Adding more granular error handling in the validate_record function became necessary to gracefully handle such parsing errors and to provide detailed debugging information, such as the line number and field content that caused the issue.

0.6.3. *Progress and Reflections.*

Despite these challenges, by the end of Day 6, I had made significant strides in my understanding of Rust's error handling and data validation mechanisms. I learned the importance of precise type handling when dealing with complex data structures and the necessity of thorough data validation to ensure the integrity of the conversion process.

Moreover, the day's work underscored the value of Apache Arrow in handling large datasets efficiently, validating my choice to use it for scalability and performance in data science tasks.

0.7. **Day 7: Structuring for Documentation and Compliance.**

On the seventh day of my data science journey with Rust, I began with the administrative task of setting up a proper documentation and task management system. This was crucial as my project grew in complexity and required a more structured approach to keep track of the various components and dependencies involved.

0.7.1. *Emphasizing Task Management.*

I started the day by creating a TODO.md file, which would act as a centralized checklist for my ongoing and upcoming tasks. This file is vital for maintaining a clear overview of my objectives and ensuring that nothing important slips through the cracks as the complexity of the project increases.

The primary item on my to-do list was to document my learning journey, a task that would not only solidify my understanding but also serve as a guide for future reference. This documentation was to be carried out using Typst, chosen for its combination of simplicity and power, ideal for a project where clarity is paramount.

```
# TODO
- [ ] Document my learning journey using Typst.
```

0.7.2. *Commitment to Legal Acknowledgement.*

Recognizing the significance of the open-source crates that empowered my project, I decided to add another task aimed at creating a NOTICE.md file. This file would acknowledge the contributions of the various crates and their authors, as well as ensuring that I adhere to the licensing requirements associated with their use.

```
Copy code
- [ ] Add a `NOTICE.md` file for crediting all crates used in the
project, with their respective licenses.
```

0.7.3. *Day's Reflection: Beyond Coding.*

Day 7 might not have been characterized by coding breakthroughs, but it was a day that highlighted the importance of the often overlooked aspects of software development: documentation and compliance. By addressing these areas, I was laying down a solid foundation for managing the project's growth and complexity.

The lessons from this day were clear—successful software development is multifaceted, with coding being just one aspect among many that include project management, documentation, and legal considerations. The groundwork laid on this day was crucial for a smooth journey ahead.

References

Management Information Systems, King Saud University, Riyadh, Saudi Arabia
Email address: salmanium.dev@gmail.com