ENAE 380: Final Project Design

Due on December 16, 2024 at 11:59 PM

Dr. Mumu Xu, 0106

Vai Srivastava

December 16, 2024

1 Overview

This document provides a detailed examination of the design and structure of the NFL Super Bowl prediction tool, which uses Rust, RSS feeds, and NLP-based question answering. The approach prioritizes a clean modular architecture, separating concerns such as data retrieval, context construction, and prediction into their own source files.
At a high level, the system fetches a series of RSS feeds related to professional football, extracts the textual context from these feeds, and then uses a natural language processing (NLP) model to answer user questions about the upcoming Super Bowl. The tool is invoked via a command-line interface (CLI), allowing users to specify a custom query and optionally enable verbose output for diagnostic insights.

2 Architecture

The project is structured into the following modules:

- `main.rs`: Entry point for the CLI, handling argument parsing and initiating the prediction flow.
- `context.rs`: Responsible for fetching RSS feed data, extracting the relevant text-based context, and returning a consolidated string that can be passed to the prediction step.
- `predict.rs`: Manages the question answering logic using the `rust-bert` library. It takes a question and context as input, returning a predicted answer.
- `rss_read.rs`: Focuses on fetching and parsing RSS feeds, returning a Polars `DataFrame` that encapsulates the relevant articles and metadata.

This modular breakdown promotes clarity and testability. Each file focuses on a distinct piece of the pipeline, making the codebase easier to reason about.

3 Data Flow and Control

When a user invokes the tool, the CLI parameters (including the question and verbosity) are parsed in `main.rs`. Following that:

1. **Context Retrieval:** Control is handed to the `context.rs` module, which orchestrates RSS retrieval via `rss_read.rs`. The result is a text-based context string derived from RSS feed descriptions.
2. **Prediction:** Once the context is acquired, `main.rs` calls into `predict.rs`, passing both the user's question and the constructed context. The `predict` function then uses a QA model to produce an answer.
3. **Output:** Finally, the answer is printed to the console.

By separating concerns, we can easily add new RSS sources, replace the NLP model, or modify the question answering pipeline without entangling unrelated components.

4 Module Overview

4.1 `main.rs`

The `main.rs` file serves as the CLI entry point. It leverages the `clap` crate to parse command-line arguments, providing a question prompt and a verbosity toggle.

4.1.1 Implementation

```rust
mod context;
mod predict;
mod rss_read;

use clap::Parser;

#[derive(Parser, Debug)]
/// Welcome to the NFL Super Bowl Predictor. You can ask a question of your
    choosing,
/// or simply use the default.
struct Args {
    /// Question to ask NFL Predictor
    #[arg(short, long, default_value_t = ("What team is most likely to win the
    upcoming Super Bowl?").to_string())]
    question: String,

    /// Verbose Mode
    #[arg(short, long, default_value_t = false)]
    verbose: bool,
}

#[tokio::main]
async fn main() {
    let args = Args::parse();

    let context_str = context::get_context_str(args.verbose).await;
    let question_str = args.question;

    predict::predict(question_str, context_str.unwrap()).await;
}
```

4.1.2 Code Explanation

- `mod context; mod predict; mod rss_read;`: Declares that these modules (located in similarly named files) are part of the crate. This makes their functions accessible.
- `use clap::Parser;`: Imports the `Parser` trait from the `clap` crate, enabling automatic command-line argument parsing from a struct definition.
- `#[derive(Parser, Debug)] ... struct Args ...`: Defines a struct `Args` that `clap` will use to parse command-line arguments. The `Debug` derive is for logging/debugging, and `Parser` is for command-line parsing.
- `#[arg(short, long, default_value_t = ...)] question: String`: Declares that `-q/--question` is an argument that defaults to the provided string if not supplied.
- `#[arg(short, long, default_value_t = false)] verbose: bool`: Declares a boolean flag `-v/-verbose` that defaults to false.
- `#[tokio::main]`: Marks `main()` as the asynchronous entry point, using the Tokio runtime.
- `let args = Args::parse();`: Invokes `clap` to parse command-line arguments and populate the `args` struct.

- `let context_str = context::get_context_str(args.verbose).await;`: Calls into `context::get_context_str`, passing the verbosity flag. This returns a `Result<String, ...>` which will provide the textual context after fetching and processing RSS feeds.
- `let question_str = args.question;`: Extracts the user's question from parsed arguments.
- `predict::predict(question_str, context_str.unwrap()).await;`: Invokes the `predict` function from the `predict.rs` module, unwrapping the `Result` to get the actual context string. It then runs the prediction and prints the answer.

4.2 `context.rs`

This module bridges RSS data retrieval and context construction. It fetches multiple RSS feeds in parallel, aggregates them using Polars, and extracts a column (e.g., descriptions) into a single textual context.
Key responsibilities:

- Interfacing with `rss_read.rs` to get an up-to-date `DataFrame` of posts.
- Converting RSS descriptions into a consolidated context string.
- Handling verbosity by printing debugging output when requested.

4.2.1 Implementation

```rust
use crate::rss_read::*;
use polars::prelude::*;

async fn get_rss(verbose: bool) -> Result<Series, Box<dyn std::error::Error>> {
    let urls = vec![
        "https://rss.nytimes.com/services/xml/rss/nyt/ProFootball.xml",
        "https://api.foxsports.com/v2/content/optimized-rss?...",
        "https://www.espn.com/espn/rss/nfl/news",
        "https://www.cbssports.com/rss/headlines/nfl/",
        "https://profootballmania.com/feed/",
        "https://feeds.washingtonpost.com/rss/rss_football-insider",
        "https://sportspyder.com/nfl/philadelphia-eagles/news.xml",
    ];

    let rss_feeds = update_feeds(urls).await;

    if verbose {
        match &rss_feeds {
            Ok(rss_feeds) => println!("{:?}", rss_feeds),
            Err(e) => eprintln!("Error: {:?}", e),
        };
    }

    let rss_df = rss_feeds.unwrap();
    let rss_col = extract_col(&rss_df, "description").await;

    if verbose {
        match &rss_col {
            Ok(rss_col) => println!("{:?}", rss_col),
            Err(e) => eprintln!("Error: {:?}", e),
        };
    }

    rss_col
}

pub async fn get_context_str(verbose: bool) -> Result<String, Box<dyn
    std::error::Error>> {
    let context_series = get_rss(verbose).await?;
    let context_str = context_series.to_string();
    Ok(context_str)
}
```

4.2.2 Code Explanation

`get_rss(verbose: bool)`

This function fetches multiple RSS feeds and extracts their descriptions as a Polars `Series`.

- `let urls = vec![...];`: A hard-coded vector of RSS feed URLs from various NFL-related sources. These will be fetched and concatenated.
- `let rss_feeds = update_feeds(urls).await;`: Calls `update_feeds` (from `rss_read.rs`) to retrieve and parse the RSS data into a `DataFrame`. This returns a `Result<DataFrame, ...>`.
- `if verbose ...`: If verbosity is enabled, we print the retrieved data or any errors that occurred.
- `let rss_df = rss_feeds.unwrap();`: Unwrap the `Result` to get the `DataFrame`. If an error occurred, the program will panic here. In a production environment, more robust error handling might be desired.
- `let rss_col = extract_col(\&rss_df, "description").await;`: Extracts the `"description"` column from the `DataFrame`. This returns a `Result<Series, ...>`.
- Another verbosity check prints details of the resulting `Series`.
- Finally, returns `rss_col`, which should contain a `Series` of all RSS descriptions combined.

`get_context_str(verbose: bool)`

This public function ties it all together and returns a `String` of RSS descriptions.

- `let context_series = get_rss(verbose).await?;`: Awaits the `get_rss` function and uses the `?` operator for error propagation.
- `let context_str = context_series.to_string();`: Converts the `Series` to a `String`, effectively concatenating all the fetched descriptions.
- `Ok(context_str)`: Returns the combined context string.

4.3 `predict.rs`

The `predict.rs` file is where the NLP logic resides. It employs the `rust-bert` library's `QuestionAnsweringModel` to generate an answer to a user-supplied question given the aggregated context.
Main operations:

- Spawns a blocking task to initialize the QA model.

- Runs a prediction using the user's question and the RSS-derived context.
- Prints out the final predicted answer.

4.3.1 Implementation

```rust
use rust_bert::pipelines::question_answering::*;
use tokio::task;

pub async fn predict(question: String, context: String) {
    let qa_model = task::spawn_blocking(|| {
        QuestionAnsweringModel::new(Default::default()).expect("Failed to create QA model")
    })
    .await
    .expect("Blocking task failed");

    let answer = qa_model.predict(
        &[QaInput {
            question: question.clone(),
            context: context.clone(),
        }],
        1,
        2048,
    );

    println!("{:?}", question);
    println!("{:?}", answer[0][0].answer);
}
```

4.3.2 Code Explanation

- `use rust_bert::pipelines::question_answering::*;`: Imports the question answering pipeline components.
- `use tokio::task;`: For running blocking model initialization in a separate thread.
- `pub async fn predict(question: String, context: String)`: The entry point that receives a user's question and the prepared context.
- `let qa_model = task::spawn_blocking(|| ...).await.expect("...");`:
  - `spawn_blocking`: Runs the provided closure on a thread pool for blocking tasks, ensuring the main async runtime isn't blocked by heavy CPU operations.
  - Inside the closure: `QuestionAnsweringModel::new(Default::default())` initializes the QA model with default settings (it will load weights and configuration files).
- `let answer = qa_model.predict(...)`: Calls the `predict` method, passing a slice of `QaInput` containing the user's question and context. We limit answers to 1 and use a maximum context length of 2048 tokens.
- `println!(..., question); println!(..., answer[0][0].answer);`: Prints the user's question and the predicted answer from the model, typically the best guess at which team might win the Super Bowl.

4.4 `rss_read.rs`

Finally, `rss_read.rs` deals with fetching and parsing RSS feeds into a Polars `DataFrame`. Here, we:

- Retrieve multiple RSS feeds concurrently using `reqwest`.
- Parse them into `DataFrame` columns such as `title`, `link`, `description`, and `pub_date`.
- Provide a helper function `extract_col` to retrieve a particular column from the `DataFrame`.

4.4.1 Implementation

```rust
use polars::prelude::*;
use reqwest::Client;
use rss::Channel;
use tokio::task;

pub async fn update_feeds(urls: Vec<&str>) -> Result<DataFrame, Box<dyn
    std::error::Error>> {
    let client = Client::new();
    let mut lazy_frames = Vec::new();

    for url in urls {
        let rss_feed = fetch_rss_feed(url, &client).await?;
        lazy_frames.push(rss_feed.lazy());
    }

    let concatenated_lazyframe = concat(lazy_frames, UnionArgs::default())?;
    let all_feeds_df = task::spawn_blocking(move ||
    concatenated_lazyframe.collect()).await??;

    Ok(all_feeds_df)
}

async fn fetch_rss_feed(
    url: &str,
    client: &Client,
) -> Result<DataFrame, Box<dyn std::error::Error>> {
    let response = client.get(url).send().await?.bytes().await?;
    let channel = Channel::read_from(&response[..])?;

    let titles: Vec<String> = channel.items().iter()
        .map(|item| item.title().unwrap_or_default().to_string())
        .collect();

    let links: Vec<String> = channel.items().iter()
        .map(|item| item.link().unwrap_or_default().to_string())
        .collect();

    let descriptions: Vec<String> = channel.items().iter()
        .map(|item| item.description().unwrap_or_default().to_string())
        .collect();

    let pub_dates: Vec<String> = channel.items().iter()
        .map(|item| item.pub_date().unwrap_or_default().to_string())
        .collect();

    let rss_posts = df![
        "title" => titles,
        "link" => links,
        "description" => descriptions.clone(),
        "pub_date" => pub_dates.clone(),
        "date_description" => pub_dates.iter()
            .zip(descriptions.iter())
            .map(|(p, d)| p.to_owned() + ": " + d.to_owned())
            .collect::<Vec<String>>(),
    ]?;

    Ok(rss_posts)
}

pub async fn extract_col(
    rss_posts: &DataFrame,
    column: &str
) -> Result<Series, Box<dyn std::error::Error>> {
    let descs_df: Series = rss_posts[column].clone();
    Ok(descs_df)
}
```

4.4.2 Code Explanation

`update_feeds(urls: Vec<&str>)`

- `let client = Client::new();`: Creates a new `reqwest` HTTP client to fetch RSS data.
- `let mut lazy_frames = Vec::new();`: Will store Polars `LazyFrame` objects for each feed.
- `for url in urls ...`: Iterates over each RSS feed URL.
  - `let rss_feed = fetch_rss_feed(url, \&client).await?;`: Asynchronously fetches and parses the RSS feed for the current `url`, returning a `DataFrame`.
  - `lazy_frames.push(rss_feed.lazy());`: Converts the `DataFrame` into a `LazyFrame` and accumulates it. LazyFrames are not immediately computed, allowing for potential optimizations.
- `let concatenated_lazyframe = concat(lazy_frames, UnionArgs::default())?;`: Concatenates all LazyFrames into one large LazyFrame. The `concat` function merges multiple data sets into a single structure.
- `let all_feeds_df = task::spawn_blocking(move || concatenated_lazyframe.collect()).await??;`:
  - Uses `spawn_blocking` again because `collect()` from Polars can be CPU-intensive.
  - The double `?` handles both the `spawn_blocking` result and the inner `Result` from `collect()`.
- `Ok(all_feeds_df)`: Returns the fully realized `DataFrame` with all combined feed data.

`fetch_rss_feed(url: &str, client: &Client)`

- `let response = client.get(url).send().await?.bytes().await?;`: Sends a GET request to `url`, awaits the response, and collects the entire response body as bytes.

- `let channel = Channel::read_from(\&response[..])?;`: Parses the RSS channel from the raw bytes using the `rss` crate.
- Constructs vectors (`titles`, `links`, `descriptions`, `pub_dates`) by iterating over `channel.items()`.
- `let rss_posts = df![ ... ]?;`: Constructs a Polars `DataFrame` from the collected vectors. The `date_description` field merges publication date and description into a single string.
- `Ok(rss_posts)`: Returns the assembled `DataFrame`.

`extract_col(rss_posts: &DataFrame, column: &str)`

- `let descs_df: Series = rss_posts[column].clone();`: Selects the specified column from the DataFrame and clones it into a `Series`.
- `Ok(descs_df)`: Returns the column as a `Series`.

5 Conclusion

During the development and experimentation phase of this project, an interesting and somewhat concerning phenomenon was observed. Specifically, when the final assembled context is handed over to the question-answering model, the latter part of the provided data appears to exert a disproportionately strong influence on the outcome. This was noted to the extent that even a team with no realistic chance of winning the upcoming Super Bowl—perhaps one that was definitively eliminated from playoff contention—could be predicted as the winner if it happened to be mentioned frequently in the last few articles appended to the context. In other words, the temporal ordering of the articles (and their final appearance in the provided context) can overly bias the model's answer.

This behavior highlights a fundamental volatility in large language models (LLMs). Despite the *pre-trained knowledge* these models carry (information encoded in their internal weights from vast swathes of text encountered during their initial training phase), the final result is still heavily shaped by the *contextual prompt knowledge*, i.e., the text provided to the model at inference time. While it is often assumed that LLMs will faithfully combine their extensive background knowledge (pre-trained into them) with the immediate prompt to produce a balanced, context-aware response, what this project shows is that the most recently seen context can overshadow earlier parts of the prompt and even run counter to the model's internal understanding.

Why does this happen? LLMs operate probabilistically, selecting the next likely token based on both their training-induced priors and the currently visible prompt. Although the model's pre-trained weights contain general knowledge (for example, about which teams remain in contention late in the NFL season and a baseline understanding of sports outcomes), the model does not have any sense of "fact-checking" or temporal reasoning that would override a more recent textual suggestion. If the final documents in the prompt repeatedly mention a particular team and associate it with success, the model—due to the recency bias in how it processes context—will often "go along" with this suggestion and produce that team as the predicted winner, regardless of the broader factual situation.

This reliance on prompt context over pre-trained knowledge becomes problematic when the provided context is misleading, incomplete, or heavily skewed towards a particular narrative. The model's architecture and decoding algorithm do not allow it to easily reject contradictory information if it is strongly emphasized near the end of the prompt. The result is a fragile and volatile set of predictions, where slight changes to the ordering or selection of articles in the prompt can drastically alter the model's answer.

For use cases like sports predictions, such volatility can undermine trust in the model's outputs. Users may expect that a model "knows" which teams are still in contention or has an inherently stable understanding of the NFL landscape. However, what this example shows is that an LLM's final output is a product of a delicate interplay between its deeply embedded, pre-trained statistical patterns and the immediate, ephemeral textual context it is given. Unless carefully managed—by pruning irrelevant context, weighting or filtering articles, or employing more advanced reasoning layers—LLMs can easily be swayed by the last few lines of text they are fed.

In short, the observed phenomenon underscores the importance of understanding the delicate and sometimes counterintuitive relationship between a model's internalized knowledge and the prompt-provided context. Such insights are crucial for building more robust, reliable, and factually grounded systems that can perform consistently across a variety of scenarios.