Brandon Lau

DS210 Sec. B

My GOAT Tom Brady

https://github.com/Brandmuffin1/ds210finalproject

My project is on Tom Brady's pass attempts during his time with the Tampa Bay Buccaneers. I chose not to test his entire career's worth of pass attempts because that is simply too much (over 12000), whereas, during his time with the Bucs, he had a little over 2400 (including playoffs). I found a complete play-by-play of every NFL game (which includes every pass, run, special teams play, change of quarter, etc) called the nflverse-pbp by nflverse on GitHub (https://github.com/nflverse/nflverse-pbp?tab=readme-ov-file). To view this, I first used Python in a Jupyter Notebook (210finalproject.ipynb) to install nfl_pbp_data_py because there were only Python and R options and I was more familiar with Python. Then, I imported just the data from the 2020, 2021, and 2022 seasons in a CSV file called nfl_pbp_2020_to_2022.csv, because those were the years that Brady played in Tampa. The steps to do so are here: https://pypi.org/project/nfl-data-py/. I added that to my Rust directory and cleaned and filtered the dataset to my desires. My project implements graph clustering and partitioning. The graph nodes represented by pass attempts and receivers allow us to analyze things like which receivers were targeted the most. The graph naturally clusters around receivers because multiple pass nodes connect to the same receivers telling us where every Tom Brady pass attempt went (in a list with 2439 edges). The number of unique receivers targeted tells us about the clustering around Tom Brady; the quarterback is the parent node and the receivers are child nodes (34 total nodes). The receiver target count portion allows us to visualize the receiver as the “parent” node and each target as a child node, which tells us which receiver was Brady’s favorite target. With

some football background knowledge, these analyses line up with what is known about Tom Brady's time with the Buccaneers. Head coach Bruce Arians is a very aggressive play-caller, as seen in the sheer amount of pass attempts in just 3 seasons (2439 in regular season and playoffs). This is also seen in Mike Evans and Chris Godwin leading the way in targets as they are the top receiving options in a vertical passing scheme (Evans as the big play receiver and Godwin as the intermediate and run-after-catch receiver).

It all starts with fn clean_csv in the cleaning.rs module. Its job is to process the raw CSV file (nfl_pbp…) and fill in any missing or blank fields with "0" to ensure all rows have the same number of columns as the header row and output an updated CSV file called filtered_nfl.csv. Here, every single column (there are a lot) is filled.

- let mut rdr uses ReaderBuilder to read the input file nfl_pbp… and flexible(true) allows the reader to handle rows of varying lengths without immediately throwing an error because the raw dataset contains rows with missing values, this ensures those rows can be processed.
- let mut wtr uses WriterBuilder to output the cleaned data to output_file.
- let headers reads the header row from the input file and clones the headers to be used in the output file. It writes the headers to the output file using wtr.write_record to ensure the output file has the same headers as the input file.
- The for loop iterates through each row in the input CSV and numbers it for debugging purposes. The match block checks whether a row was successfully read, and processes it, or skips it, and throws an error message. This prevents the program from crashing if a row is invalid.

- let cleaned_row ensures that each row has the same number of columns as the header. For each column, it retrieves the corresponding value in the row using record_get(i). If the value doesn't exist, it uses an empty string and then replaces it with "0".
- Err(err) prints an error message for rows that couldn't be processed for debugging purposes.
- Wtr.flush ensures that all data is written to the output file by flushing the writer's buffer and returns Ok(()) to indicate the operation is successful.

We then move to fn filter_tom_brady in the filtering.rs module which processes the filtered_nfl.csv file and filters just Tom Brady's ("T.Brady") pass attempts ("pass").

- Mut rdr, mut wtr, headers, the for loop, match block, Err(err), and wtr.flush have the same function as the ones from clean_csv.
- The if record.len… statement checks if the number of columns in the row matches the header row. It throws an error and skips the row if the lengths are unequal. This prevents mismatched rows from causing trouble.
- let passer finds the indices of the passer using headers.iter().position and extracts the corresponding values from the row using record_get(index). It defaults to an empty string if the column is missing.
- let play_type does the same but with passes.
- If passer… checks if the row satisfies the criteria that "T.Brady" is the passer and the play type is "pass". If the criteria are met, then it's written to the output CSV tom_brady_pass_attempts.csv.

Next is select_columns in the selecting.rs module. It, once again, takes the previous CSV created (tom_brady…) reads it, and writes a new file (brady_filtered_columns.csv).

- let desired_columns defines the list of column names that should be retained in the output file. It ensures that only the desired columns are included, reducing the dataset size.
- let headers and let indices maps reads the header row of the input file to identify column indices corresponding to the desired columns. Filter_map skips columns that don't exist in the input file.
- let selected_headers retrieves the header names for the desired columns using the mapped indices
- The for loop has the same function as previously.

To build the graphs, we use fn build_graph_from_csv in the graph.rs module. This fn builds a directed graph (ReceiverGraph) from the CSV brady_filtered… Each pass attempt is a node and each receiver becomes another node and the directed edges connect pass attempts to their respective receivers.

- We use let mud rdr to read the file. Let mut graph to initialize the directed graph ReceiverGraph. Let mut receiver_nodes uses a HashMap to track receiver names and their corresponding graph nodes (NodeIndex) to avoid duplicate nodes for the same receiver.
- The for loop loops through each row in the file and provides the row number for debugging. It extracts the receiver field (column 6) and defaults to an empty string if the field is missing. It gives a warning and skips rows where the receiver field is blank.
- Let receiver_index uses a HashMap to check if the receiver node already exists. If not, it creates a new node in the graph and stores its NodeIndex in the map.

- Let pass_attempt_vertex and pass_attempt_index creates a unique node for each pass attempt using the row number. It connects the pass node to the corresponding receiver node with a directed edge (pass → receiver).

To see the graph, we call fn analyze_graph which prints the nodes and edges and lists the connections between passes and receivers. It also counts passes that had no intended receivers such as spikes, throwaways, penalties, etc. It also counts up each target for each unique receiver.

- The print statements at the top of the fn print the total number of nodes in the graph (pass attempts + unique receivers) and the total number of edges (pass → receiver).
- The for edge for loop iterates over all the edges in the graph and extracts the source (pass node) and target (receiver node) for each edge. If the receiver is “0”, it marks the pass as “Incomplete/Penalty” (although it could be a number of different results as aforementioned) and prints the total.
- Let unique_receivers isolates each unique receiver node and prints out the total number of distinct receivers who earned a target from Tom Brady.
- Let mut receiver_counts loops through the receiver nodes and counts the number of edges (targets) for each receiver and stores it in a HashMap.
- Let mut sorted_counts converts the HashMap into a vector of (receiver, count) pairs and sorts it by descending order.