Natalie Suh

Final Project Report

1. **What the Project Does**
    a. In short, this project calculates average statistics for each NHL team in a given year. It builds and displays a graph representing relationships between teams based on statistical similarity. It also identifies the nearest neighbors (teams with the most similar statistics) for a given year and team using k-nearest neighbors.
    b. The program uses a CSV file entitled game_teams_stats.csv, which contains NHL team game statistics from all games between 2000 and 2019.
        i. The game statistics being analyzed include the following: goals per game, shots per game, hits per game, penalty minutes per game, power-play opportunities per game, power-play goals per game, faceoff win percentage per game, giveaways per game, takeaways per game, and blocked shots per game.
2. **How to Download the Data**
    a. The CSV necessary for this project is included in the zip file available for download from GitHub. If you encounter any issues accessing it, you can also find it here: https://drive.google.com/drive/folders/1_CNKMllN2ZKcj0nWDH4BsvcpJQZR7iwb?usp=sharing
3. **Code Explanation**
    a. The code I wrote for this project is split into six rs files, which I will explain one at a time.
    b. data.rs
        i. My code for data.rs begins with my imports. I imported ReaderBuilder, which is used to create a CSV reader with configurable options, such as whether the file has headers. I also imported ndarray, which allows me to store numerical data in a one-dimensional array. I also imported HashMap, which allows me to map the team IDs to their statistics averages and store them in hashmaps. Finally, I imported the trait object error, which helps handle errors generically.
        ii. Next, I defined a struct that represents my data-loading logic. It holds the file path for my CSV file and the target year (given by the user) for filtering data. I made sure this struct was public, so that it could be accessed across files.
        iii. After that I write out the implementation of DataLoader, which includes the function new. This constructor helps create a new dataloader, converting our file path and year from string slices into owned string values.
        iv. I also write out a function load_and_average, which reads the CSV file and returns a result. On success, it will return a hashmap mapping team IDs to their averaged stats, and on error it will return a boxed error for general error handling.
            1. It creates a new csv::ReaderBuilder, which is used to configure CSV file reading. It indicates that the CSV file has a header row and opens the file specified in the file_path. The ? indicator propagates any error that occurs when opening or parsing the file.
            2. Next, it reads the header row from the CSV file and returns the headers as a record, creating a clone for use later without modifying the original.

3. After this, it initializes the column indices, storing the index positions of the team and game id columns in option variables. It also creates a vector "numeric_columns" that stores the indices of columns containing numeric data.
4. Next, it identifies relevant columns using a for loop. It iterates over the headers, providing the index and header name. It checks that the header matches team_id and if true, assigns the current index to team_id_idx. If not team_id, it checks for game_id and assigns the index to game_id_idx. Otherwise, it identifies the numeric columns by ensuring all characters in the header are alphanumeric or underscores. Finally, it adds the index of numeric columns to numeric_columns.
5. After this, it validates columns and prepares to return an error message if the team_id or game_id are not found.
6. It initializes data storage, creating the hashmap. It then iterates over the rows in the CSV file, returning each row as a result. It unwraps the row, propagating errors if parsing fails.
7. After this, it filters by year, extracting the first four characters (which represents the year) from the game id and skips rows where the year doesn't match the given year.
8. It also extracts the team id and converts it to a string. It iterates over the indices of numeric columns and attempts to parse each value as a float. It defaults to a 0.0 if parsing fails. And collects the parsed values into a vector. It groups the rows by team ID, retrieving existing entries from team_id or inserting a new empty vector. It then adds the parsed numeric data as an array to the vector.
9. It computes the team averages, by calculating the number of rows for that team, summing the rows and dividing the summed vector by the number of rows. It collects the averages into a hashmap. At the end it returns the averages, wrapped in a result::ok.

v. This file also includes tests, which test both functions. It includes necessary imports. The first test verifies that the DataLoader constructor initializes correctly. The second creates a temporary CSV, writes mock data, and verifies that load_and_average computes the correct averages.

c. graph.rs
   i. This code starts with my imports. I import petgraph to allow me to represent a graph structure with nodes and edges. I also import undirected, which specifies that the graph is undirected. I import the hashmap and hashset collections and the ndarray for creating one-dimensional arrays.
   ii. I start by creating a function to calculate the Euclidean distance between two vectors. It iterates over paired elements, computes squared differences, sums them, and takes the square root. This will be useful for creating our graphs.
   iii. Next comes my function for creating a graph, which is public so that it can be used in other files. It takes a hashmap of team averages and outputs a graph and hashmap of team IDs to node indices.

1. I initialize an undirected graph where nodes are teams and edges are relationships between teams. I map the team IDs to the corresponding node indices in the graph. I also set a threshold, which defines the maximum Euclidean distance between two teams for an edge to be created, which allows the graph to be more sparse and meaningful.
2. I filter out teams whose sum of statistics is less than or equal to 10.0, which further declutters the graph. It stores these team IDs in a hashset.
3. I then add nodes, iterating over the averages and adding nodes to the graph for each team in filtered teams. I map the team ID to the corresponding node index.
4. I add edges, using a nested loop to iterate over each team and compare it to other teams. It ensures both are distinct and in filtered_teams. It computes euclidean distance and adds an edge between the nodes if the distance is less than the threshold.
5. This code will return a constructed graph and the mapping of team IDs to node indices.

iv. I also wrote a test for create_graph, which creates mock data and verifies that create_graph results in the correct number of nodes and team IDs in team_indices.

d. knn.rs

i. For knn.rs, I begin with importing ndarray and hashmap.

ii. I start by creating a function to calculate the Euclidean distance between two vectors. It iterates over paired elements, computes squared differences, sums them, and takes the square root. This will be useful for doing our KNN analysis. It is pretty similar to the one in the graph file.

iii. I create a function for KNN, which will find the k nearest neighbors for a given query point in a dataset. It will return a vector of tuples containing the index of each neighbor and its distance to the query.
1. I start with the distance calculation, which iterates over rows of the dataset, calculating the Euclidean distance to the query point and stores the row index and distance in a vector.
2. I then sort the distances in ascending order and take the k smallest distances, returning them.

iv. After this general KNN function, I write a function to find the k nearest teams for each team in team_averages. It will return a hashmap, where the keys are team IDs and the values are vectors of nearest neighbors including the ID and distance.
1. I collect all team IDs into a vector and convert my team averages into a matrix. Each row corresponds to a team's averaged statistics.
2. I iterate over the row, retrieving each team's row from the matrix. I call KNN to find the k+1 nearest neighbors (which helps exclude the team itself from being returned) and truncate to k neighbors. I ensure the neighbors are sorted by distance.

3. I then build a result map, which converts neighbor indices back to team IDs and distances and collect the team-to-neighbor mappings into a HashMap.

v. I also wrote tests for this to ensure that my functions were working properly. The first test creates a dataset of 3 points and a query point. It verifies that knn returns 2 neighbors and that the distances are sorted. The second test creates a mock team_averages and verifies that find_nearest_teams correctly finds the 2 nearest neighbors for each team.

e. visualization.rs

i. For visualization.rs, I start with my imports. I import plotters, which helps to create visualizations. I also import petgraph, which represents the graph structure used for visualizing team interactions and provides functionality for iterating over graph edges. Finally, I import error to handle errors generically.

ii. This file contains the function draw_graph, which takes a graph and output path for the visualization and produces a PNG image of the graph.

1. I use BitMapBackend to create an 800x800 pixel canvas and set the background color to white.
2. I then configure the chart's layout, including a title and margin. I use a 2D cartesian coordinate system for my chart via ChartBuilder.
3. I then calculate the node positions by collecting the node indices, positioning them evenly around a circle based on their index and total number of nodes, and store positions as coordinates in positions.
4. After this, I iterate over the graph edges, retrieve the positions of source and target nodes, and draw a line using LineSeries between two nodes if their positions exist in positions. This helps draw my edges.
5. I draw my nodes, using PointSeries to draw a red circle at the correct position.
6. I add my labels, calculating the position for the label slightly outside the node and place it down using the node's name.
7. I then finalize the chart, saving it to the output path, and return Ok(()) on success.

f. main.rs

i. For visualization.rs, I start with my imports. I first import my modules, which allows me to access the structs, methods, and functions I described above. I also import error for handling errors and io for providing input/output functionality.

ii. After this comes my main function, which is the entry point of the application and returns a result signifying successful execution.

1. The first thing my main function does is import my csv, which is hard coded as "game_teams_stats.csv."
2. After that, it prompts the user to enter a year for analysis. It reads the input into year and trims extra whitespace.
3. It then loads the data. It instantiates a DataLoader object with the CSV file path and input year. It then calls load_and_average to compute averages for each team based on the filtered data.

4. After this, it checks for empty data. If no data matches the specified year, the program prints a message and exits gracefully.
5. It then calls create_graph to build a graph from team_averages. I left team_indices unused here, but it could potentially be used to map team IDs to graph nodes. It also calls draw_graph to save the graph visualization as a PNG file and notifies the user upon a successful save.
6. Then, it calls find_nearest_teams to compute the 3 nearest neighbors for each team in team_averages.
7. It prompts the user to enter a team ID for which they want to view the nearest neighbors, reading and trimming the input before storing it in selected_team.
8. After this, it looks up the entered team in the neighbors map. If found, it displayed the nearest neighbors and their distances. If not, it prints an error message.
9. It exits the program successfully if all operations complete without errors.

4. How to Run the Code

a. To run the code, **download the zip file from GitHub**. Ensure that you have a rust project with the proper files, including: the CSV file game_teams_stats.csv in your project root; a src directory containing data.rs, graph.rs, knn.rs, visualization.rs, and main.rs; and the proper dependencies in your cargo.toml file (csv = "1.1", ndarray = "0.15", petgraph = "0.6", plotters = "0.3", tempfile = "3.3").

b. Then, **compile and run using cargo build and cargo run**. You also have **the option to test the functions, using cargo test**.

c. You will be prompted to **enter a year**. The year that you choose will determine what game data will be utilized for analysis (so choosing 2016 will mean analyzing all data from games in 2016). **Enter any year from 2000-2019 following the format suggested** (i.e. 2016).

   i. As long as the year is in the above range, the program should proceed.

d. After this, you will be prompted to **enter a team ID**, which means entering a team name. You may **choose any NHL team from this list, following the written format:** Devils, Flyers, Kings, Lightning, Bruins, Rangers, Penguins, Red Wings, Sharks, Predators, Canucks, Blackhawks, Senators, Canadiens, Wild, Capitals, Blues, Ducks, Coyotes, Islanders, Maple Leafs, Panthers, Sabres, Flames, Avalanche, Stars, Blue Jackets, Jets, Oilers, Golden Knights, Hurricanes, Coyotes, or Thrashers.

   i. The team that you choose will determine what team data will be utilized for analysis (so choosing the Bruins will mean analyzing all data from Bruin's games in the given year).

   ii. You shouldn't run into any issues if choosing from the above list. **However, if the team you selected does not yield results, try another**. I suggest the Bruins (yay Boston teams!!).

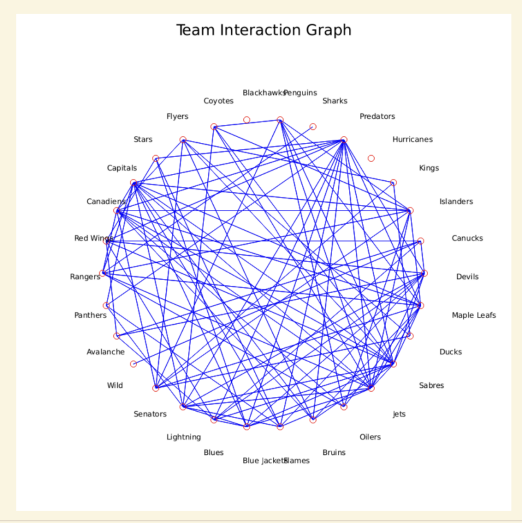
5. Expected Output

a.

```
[/opt/app-root/src/project]
$ cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.06s
     Running `target/debug/project`
Enter the year for analysis (e.g., 2016):
2016
Graph visualization saved as team_graph.png
Enter a team ID to see its nearest neighbors (e.g., Bruins):
Bruins
Team: Bruins -> Nearest Neighbors:
Neighbor: Flyers, Distance: 4.2498
Neighbor: Maple Leafs, Distance: 4.3650
Neighbor: Predators, Distance: 4.5043
[/opt/app-root/src/project]
$
```

i. After building and running the code and prompting the program with a year and a team to analyze, you should expect an output like the one above.

ii. It will tell you that a graph visualization has been created for the given year. Navigate to the visualization, which should have popped up in your project folder sidebar under team_graph.png and will look something like the below image.

1. 

2. The graph shows clusters/connections between teams based on how similar their performance metrics are. Teams with connections are statistically similar, while teams with no or few connections are statistical outliers. I did make use of a threshold value, which means teams have to be more similar in order to be connected on the graph.

iii. The program will also conduct a k-nearest neighbors test on the given team for the given year. It is expected to return the team name and an indication that the nearest neighbors have been returned. It will then display the team's three nearest neighbors (teams with the most similar average statistics) in the given year.

1. For example, the 2016 Boston Bruins had average statistics that were the most similar to the 2016 Philadelphia Flyers, 2016 Toronto Maple Leafs, and 2016 Nashville Predators.
2. It will also return a number next to each neighbor, which represents the Euclidean distance between two teams' average statistical performance vectors.

6. Why Does this Output Matter

a. The project output is useful because it provides data-driven insights into team performance patterns. It helps identify clusters of statistically similar teams, spot outliers, and understand performance similarities. The nearest neighbor analysis further reveals which teams are most comparable, adiding in competitive benchmarking, strategy development, and performance improvement plans. It is a useful approach for sports analysts, coaches, and researchers seeking insight from season data.
b. The output can be used in various ways, one of which is comparing a team and its nearest neighbors based on their performance metrics during the given season. By analyzing the similarities in their stats and correlating them with the teams' actual outcomes for the season, we can assess the extent to which statistical performance impacts season success. This helps reveal how much statistics influence a team's competitive standing and achievements.