


## Administrative

- Team Name/Number: Group 69
- Team Members: John Ziska (kingcoco42), Victor Qiu (VictorQ), Luke Phommachanh (lukephommachanh)
- GitHub Link: <https://github.com/kingcoco42/Project-3/tree/main>
- Video Link:  COP3530 Team 69 NBA Neighborhoods

## Proposal

- Problem: Our group is attempting to find the most similar players to a given NBA player based on different aspects of their play (e.g. scoring effectiveness and efficiency, playstyle, defensive contribution) using k-nearest neighbors and approximate nearest neighbors analyses.
- Motivation: Although NBA players can be analyzed with moderate effectiveness visually (i.e. the average viewer with no previous NBA knowledge could probably pick out the All-Star caliber players from the pack while watching a game), our analysis strives to look under the hood, so to speak. An accurate view of a player's impact on an NBA team is essential to coaching and front office staff, as a team's success can be significantly bolstered by acquiring positive contributors for cheap - or, inversely, hindered by employing mediocre or negative contributors on massive contracts that eat away at the team's limited salary cap. An effective nearest-neighbors analysis, then, could allow teams to find players who fit the same role as desirable but unattainable or impractical targets. For instance, a team may want to employ a defensive stopper like Kawhi Leonard, but be unable to afford his contract or unwilling to manage his injury concerns, but could turn to a healthier, cheaper player with a similar defensive profile.
- Features Implemented: Our implementation allows users to find the k-nearest neighbors or approximate nearest neighbors for individual NBA players, either overall or in a given season. These lists can be compiled using a variety of feature groups, including scoring, defense, impact, and style, each including relevant statistics for that aspect of the game. For instance, the scoring feature group includes points per game, field goal percentage, three-point shooting percentage, among other metrics. The UI implemented an autofill for the names to allow easier search.
- Description of Data: The data used was gathered from an NBA Statistics Repository on GitHub (<https://github.com/Brescou/NBA-dataset-stats-player-team>), combining several different regular-season statistical datasets (traditional, advanced, defensive stats, etc.) into one csv file. The dataset contains statistics for every player in the league from the 1997-98 season until the 2022-23 season.
  - Note: Several advanced metrics (e.g. defensive win shares) did not come into existence until the late 1990s or later, meaning that going back further would result in many empty values in the dataset that would need to be normalized, leading to increased inaccuracy in analysis.

- Tools/Languages/APIs/Libraries: The algorithms and main method were implemented using Python. Pandas was used to manipulate the csv file, numpy and sklearn were used for several intermediate calculations (e.g. euclidean distance, standardization), and heapq was used to implement the KD tree as a heap. The data itself came from a GitHub repository with NBA statistics (<https://github.com/Brescou/NBA-dataset-stats-player-team>). The UI uses flask, flask-CORS, and React for the frontend, and JavaScript for the backend.
- Algorithms/Data Structures Implemented: We employed two algorithms, an exact k-nearest neighbors algorithm optimized using a KD tree and an approximate nearest neighbors algorithm, both of which are implemented as their own classes in Python. A Python list is used to store the neighbors in the KD tree, while a set is used for the approximate nearest neighbors.
- Roles: Luke was responsible for the data manipulation and algorithms. John and Victor handled the user interface and front end.

### Analysis - Changes

Our actual implementation did not change much from our original plan, although our roles shifted slightly. Our UI turned out to be the most time-consuming part of our implementation, as none of us are particularly experienced with front-end design, so Luke took over the back end algorithm design, while John and Victor developed the UI and front end.

### Analysis - Time Complexity

- Data Processing (NBAPlayerSimilarity.load\_data)
  - Worst Case:  $O(n \cdot \log n + n \cdot d \cdot g)$ ,  $n$  = # of players,  $d$  = # of dimensions (features),  $g$  = # of feature groups
  - Explanation: Reading the CSV is  $O(n)$ , where  $n$  = the number of players, since each player occupies a row. The pandas data operations (e.g. groupby) are  $O(n \log n)$ , meaning that the  $O(n)$  CSV reading gets dominated in the overall complexity.
- KD Tree Construction (KDTree.build)
  - Worst Case:  $O(n^2)$ ,  $n$  = # of players
  - Explanation: The KD tree is constructed by splitting the data, point by point, along the median of a given dimension. If the data is already sorted, the KD tree “degenerates” to a linked list, causing  $O(n^2)$  worst case time complexity.
- KD Tree Search (KDTree.find\_nearest\_neighbors)
  - Worst Case:  $O(n \log k)$ ,  $n$  = # of players,  $k$  = # of nearest neighbors desired
  - Explanation: In the worst case (as described earlier) the KD tree resembles a linked list. Thus in the worst case for a search, all  $n$  nodes must be visited. At each node, the appropriate heap operation is  $O(\log k)$ . The output is sorted at the end, which is  $O(k \log k)$  (Python efficiently sorts the  $k$  neighbors) in the worst case, but since  $n$  must be  $\geq k$  (i.e. the number of neighbors desired cannot

exceed the number of players in the dataset), we can effectively ignore the  $k \log k$  in the overall time complexity. The worst case overall complexity is thus  $O(n \log k)$ .

- Approximate Nearest Neighbors Hashing (`ANN.build_index`)
  - Worst Case:  $O(n * d * t)$ ,  $n$  = # of players,  $d$  = # of dimensions,  $t$  = # of hash tables
  - Explanation: The `_hash` method called by the `build_index` method assigns each player a 0 or 1 value in each dimension, which is an  $O(1)$  operation. This results in  $n$  players getting assigned a value in  $d$  dimensions for  $t$  tables, resulting in an overall time complexity of  $O(n * d * t)$ .
- ANN Search (`ANN.query`)
  - Worst Case:  $O(d*t + n*d + k \log k)$ ,  $n$  = # of players,  $t$  = # of tables,  $d$  = # of dimensions
  - Explanation: The worst case occurs when all of the players are hashed to the same bucket, resulting in  $n$  collisions across  $t$  tables. The queried player (i.e. the player whose neighbors we are trying to find) must always be hashed in every dimension for every table, which is  $O(d * t)$ , as the hashing is  $O(1)$  for  $d$  dimensions in  $t$  tables. In order to differentiate between the other players (who in the worst case have all been put in the same bucket), we must calculate the Euclidean distance from each of the  $n$  players to the target player in each of the  $d$  dimensions of interest. Like in the KD tree, the output is sorted at the end, which again is  $O(k \log k)$ . Thus the overall time complexity is  $O(d*t + n*d + k * \log k)$ , although practically speaking  $k \log k$  is largely irrelevant since  $n \geq k$  and  $\log k$  is presumably  $< d$ , but this is not strictly the case.
- Overall Similarity Search (`NBAPlayerSimilarity.find_similar_players`)
  - Exact k-Nearest Neighbors:  $O(n * \log k + k*n)$ ,  $n$  = # of players,  $k$  = # of neighbors desired
    - Explanation: The KD tree search complexity is the same. The additional  $k*n$  term comes from looping through up to  $2*k$  candidate neighbors and performing the  $O(n)$  lookup of the player id in the `player_info` dataframe. All other operations are  $O(1)$ , aside from the loop through the `extra_results` to ensure that the desired  $k$  is met, although this will run for at most as many times as the original loop (in the case that no matches are found in the original results). Thus the overall time complexity is the complexity of the KD tree nearest neighbors algorithm plus the bigger loop:  $O(n*\log k + k*n)$ .
  - Approximate Nearest Neighbors:  $O(d * t + n * d + k \log k + k*n)$ ,  $d$  = # of dimensions,  $t$  = # of tables
    - Explanation: The approximate nearest neighbors algorithm complexity stays the same. The looping takes the same complexity as in the KD tree

option, adding on the  $k*n$  term. This is likely to simplify to  $O(n*d + k*n)$ , as  $n \geq k$  and thus  $n \geq \log k$ , and  $n$  is likely much larger than  $t$ , although for some cases or different datasets this may not be the case.

- UI Time Complexity
  - React frontend (App.jsx)
    - Worst Case:  $O(k*g + p)$ ,  $k$  = # of neighbors desired,  $g$  = # of feature groups,  $p$  = # of unique players.
    - Explanation: The React frontend is tasked with triggering the backend processes, and doesn't actually execute those processes. It is only responsible for taking in user inputs which is  $O(1)$ , and then rendering the results table which is the product of the number of neighbors desired and the the number of feature groups involved in the selection, hence  $O(k*g)$ , then there was also the autofill option for player names, and that renders and lists all the unique player names so its  $O(p)$ .
  - Backend (flask\_app.py)
    - Worst Case:  $O(n * f + k * f)$ ,  $n$  = total # of player-season records,  $f$  = # features in selected profile,  $k$  = linear-in- $k$  operations.
    - Explanation: The backend was dominated by the NBA player similarity find function. The exact search (KD-Tree) with full backtracking or degraded ANN, the system compares the target player against all  $n$  records, each with  $f$  features, yielding  $O(n * f)$  time. Furthermore, costs from formatting the top  $k$  results ( $O(k * f)$ ), and performing auxiliary operations like filtering, calling `.unique()` to extract all player names, and sorting them alphabetically ( $O(n \log n)$ ) at most, but dominated by  $(n * f)$ .

## Reflection

Overall, we had a reasonably smooth process with this project. We were able to work effectively together asynchronously, despite having slightly different schedules. Communication was clear and efficient, and we all had a shared vision for the project. Willingness to contribute existed throughout the group, and our individual talents seemed to find their niches. Our work was not without some bumps in the road, however.

Our biggest hurdle was front-end design, as none of us had extensive prior experience with it. The algorithm design and implementation went fairly smoothly, but by far the most time-consuming part of our project was producing a functional user interface that looked how we wanted it to. Part of the issue was making the algorithms (implemented in Python) compatible with the UI, which used the JavaScript-based front-end framework React.

If we were to start again, we would likely have devoted more time and manpower as we were available to the user interface design, perhaps even doing that first. The algorithms could then be tailored to fit a specific input format that would work easily with the front end. It likely would have also been helpful to push more intermediate versions of our code to GitHub for more

collaborative work - we tended, in this project, to complete functional versions of our code individually and then push to GitHub.

(What I Learned - Luke) I think the biggest leap that I made in this project was more effectively documenting my code in order to work with others. In retrospect, I have had a tendency to write code that is essentially unmaintainable by others, with variable names that make sense to me but have little inherent meaning. Having to push my work to GitHub and have it be seen and effectively used by others pushed me to have more readable, well-commented and documented code.

(What I Learned - John) I learned the value of interdependence with my group. In the past, when I had to do group projects, I would be frustrated because I felt like it would be better if I could work separately. With this project, that wasn't the case. Luke had a strong understanding of the algorithms implemented and took the lead in implementing them in code. Victor did not have experience with UI tools but was determined to learn and write most of the code for the UI. I had experience with GitHub because I used it for its timeline capabilities for the previous projects. Therefore, I set it up and delegated a few simple tasks and deadlines for our group. I am grateful GitHub was required and will use its functionality in future projects.

(What I Learned - Victor) I learned the wonders of React and frontend development. Once I got past the initial learning curve, it became a game of what else can I add. I also learned to better comment and document my programs to facilitate collaboration. Looking back, I would have started the UI development much earlier as I had little html and css knowledge with absolutely no javascript experience.

References:

K-Nearest Neighbors: <https://www.geeksforgeeks.org/machine-learning/k-nearest-neighbours/>

Approximate Nearest Neighbors:

<https://www.geeksforgeeks.org/machine-learning/approximate-nearest-neighbor-ann-search/>

KD Trees: <https://opensa-server.cs.vt.edu/ODSA/Books/CS3/html/KDtree.html>

Dataset: [Original Data Repository](#)