CSC111 Project Report: Blackjack: a Computational Analysis

Sasha Kamal Chugani Samtani, Karyna Lim, Rachel Kim, Alessia Ruberto

April 3rd, 2023

Introduction

Black Jack is a card game where players hope to get the value of their hand as close to twenty-one as possible without surpassing it. At a surface level, it seems as though winning is mostly random. While this is true due to the nature of the game, we wanted to experiment with different strategies to see if we could improve the win rate. Going into this project, we wanted to see if there was a way to optimize Black Jack and find different strategies that would work to give you the best chance at winning the hand being dealt. Black Jack at a surface level seems as though winning is mostly random. While there is definitely a large element of randomness at play, we discovered that depending on the value of the cards the player can see, there are several strategies that can be used to get closer to beating the dealer. **Our overarching goal was to create a game of Black Jack in python and utilize these strategies to optimize the player's chances at winning, then generate our own dataset of wins/losses to model the effectiveness of our strategic approach.**

We followed through with this approach, only modifying the complexity of our input parameters. For example, we decided we wanted a parameter to determine a busting threshold, which is the maximum probability that a user is willing to still hit, given the likelihood they will bust from all possible available cards from the deck. We followed through with our modifications to the target number (i.e. the number that the player is aiming to get as their sum of card values). We also increased the complexity of data analysis more than we had originally planned, comparing various configurations and finding trends in win rates between different parameter values.

Datasets

We have 2 main large datasets which we have generated ourselves to analyze factors affecting the win rate: a large original dataset and a large varied (experimental) dataset (named "large_dataframe_original_1000_games" and "large_dataframe_experimental_500_games" respectively).

The original dataset is reflective of the original Black Jack rules wherein the player is aiming to get 21 cards (i.e. target = 21) but with various game strategy configurations. For example, the original dataset has the results for games with a basic game strategy and multiple probability tree strategies with assorted threshold values.

Similarly, the varied (experimental) dataset follows the same format as the original except with experimental rules. This means we tested the game with assorted target values to see if there was a correlation between the target number and the win rate in conjunction with different strategy configurations.

Both data sets are in a CSV file format, containing the configuration of the game (threshold, target number, and strategy name) and the number of player wins, dealer wins, and pushes (ties). The CSV files themselves are very small because we pre-counted the results before adding them to the file, however, they contain the information for a very large amount of games. We chose 1000 games and 500 games respectively due to the complexity of the functions; generating each function took roughly 10 minutes and was generated using functions in the "Generating, Loading, and Exporting CSV data" section of the stats.py file.

Computational Overview

Introducing blackjack rules and its original algorithm:

Vocabulary:
- Players and dealers will be referenced as participants.
- Bust signifies going over the target value for the sum of cards to win.

Blackjack is a card game which uses the entire deck for the game where all cards represent their exact values, except for ace, queen, king, and jack. The queen, king, and jack cards are all valued at 10, and ace's value is either 1 or 11, depending on which number, added to the sum of the participant, makes them better off relative to reaching the target value (Bicycle Cards, 2022). At each round/game, the player and dealer are each dealt 2 cards, and one of the dealer's cards are faced-up. Then, the player's options for moves are either hit, draw another card from the deck, or stand, letting the dealer make their moves. The original strategy for the player is that they will hit if their current sum of cards is less than the target and if their sum is less than the dealer's face-up card plus ten. Otherwise, they will stand since they either hit the target sum needed to not lose or, given the logic, hitting would result in a bust since they're always assuming the player is closer to the target. Next, for the dealer, their logic is also dependent on the player, since their cards are revealed after their turn. The dealer only hits if the player's sum is less than or equal to the target and if their sum is less than or equal to the target minus four (originally, 17). Otherwise, they would stand, since they would win automatically if the player busts or because the likeliness of drawing a winning card(s) is higher in this basic strategy.

The data used in our blackjack implementation is based on a given deck of cards, the game state, and all possible combinations of cards and sums that could occur in the game. The results and observations of our data are all relative to the input parameters of the target sum (total value of cards to win) and/or the busting threshold, which is a maximum probability that a user is willing to still hit, given the likelihood they will bust from all possible available cards from the deck.

The four main objects in our Black Jack implementation are the participants (dealer and player), blackjack (game class), the deck, and the blackjack probability tree.

First, the deck is a dictionary mapping of all available card objects organized by a key card-type, a string, and a list of those cards as the value, where each card object contains information on its value, suit, and card type. With respect to the ace values, its value contains both possibilities and will be generated when a blackjack game is running.
- __init__() : A new deck is initialized by calling a private method, _load_standard_deck(), which generates a deck based on a mapping of card name types to their value(s), and a list of possible suits, and utilizing a nested for loop.
- __len__() : Returns an integer of the total sum of cards in the deck object. It must loop through all available card types and sum up the lengths of each associated pair list.
- draw_card(): Given a deck and a participant, randomly choose a card type and card object from the deck, remove that card from its associated card type list, and return that card to the environment. If the card type is an ace, determine its value depending on which value, added to the participant's sum of cards, would make it better off to be equal to or less than the target of 21. Additionally, the participant's sum_cards, sum value of all cards in their hand, and new_cards, list of all drawn cards, will account for the drawn card by appending the list of cards and adding that value to its previous sum. Last, if all card objects in the list of a card type are exhausted, that key in the dictionary is removed to show that this card is no longer available to draw from.
- copy_deck_and_draw_specific_card(): Return a copy of the given deck object with a specific card_type removed. This function is indifferent towards which card of that card type was chosen, as this function is essential for generating the probability tree of all possible game states after a player decides to hit or stand. It is implemented similarly to draw_card() but no player needs to be mutated, as this computes a hypothetical deck scenario. It does, however, follow the same logic as draw_cards where if a card type list is empty, it will be removed from the deck. If a card_type that's no longer in the deck is input, an error is raised as the probability tree should not account for cards that are not possible to choose from.

The Participant is a parent class where the child classes of Player and Dealer inherit the parent's initializing method, as all their attributes of the list of cards, the two initial cards drawn from the deck, and the current sum of the card values from the list must all be documented. Dealer and Player also inherit identical methods of how they manage their cards based on the current game state and for the player, which strategy they are following during the game.
- hit_or_stand(): Participant method which returns a string, which signals and computes which move the Player or Dealer should make, given the original logic of the Black Jack game previously described. For the Player, this function is only called by the blackjack object game state for the original/basic Black Jack strategy where a probability

tree is not utilized.

Next, Black Jack is the game class which initializes a starting Deck, Player, and Dealer for the game, but also registers a game state status as a string to keep track of whether the Player, Dealer or Black Jack game should be computing their next move.
- run_game(): Given an input target value as each participant's goal sum value of cards and a Black Jack game object that has been initialized with all starting attributes, play a round of blackjack by calling the specific deck functions which draw cards for each participant, and where helper functions that handle Player, Dealer, and Black Jack turns determined by which cards where drawn by each participant. This method returns the final result of the game, as a string, and which player(s), if any, won.
- run_probability_game(): Identical algorithm to run_game, yet the player's moves are handled depending on the float threshold, an additional parameter, and which move is computed by the probability tree, which determines its values on the current deck of the game state. Since this deck is updated at each hit from a participant, the probability tree recurses down to the node which represents the sum it actually received after drawing a new card.

Last, the ProbabilityTree is the tree representation of all possible card combinations a Player could obtain given their deck, the available cards to choose from, and the player's current game state of cards in their hand. Each node in this probability tree contains attributes on the current sum of cards and its subtrees which are determined by the available card types from the deck that could be added to that node's sum. For the nodes in the subtree that are computed as hypothetical scenarios of the player drawing that card (ie. not the root that the tree was originally computed from), those nodes will also include a remaining attribute that represents the amount of cards of that card type that were possible to draw right before the player hit and followed down that specific subtree's node. This probability tree is essential in determining how the player will choose which move to perform, since the proportion of receiving a card that would cause the player to bust, given by the attributes of each node, and it would be compared to its percentage threshold that the player allows busting to occur. Last, the probability tree's subtrees were implemented similarly to the deck format which highlights the relation between the cards of the deck and which cards provide a favorable move.
- generate_tree(): Given a deck and an initialized tree whose root value is a node of the current sum of the player's game state, recursively compute all possible card sums that could be drawn with respect to the current deck of the game. Probability trees with a sum that would cause the player to bust if they were to hit again, either obtaining the target value or above, have no subtrees since the player would stand in such a situation and would not hit given the nature of the game. The Probability Trees that are less than the target value would keep hitting until that sum is met, meaning at each recursive step, a copy of the original input deck is given with that specific card type removed (since that tree node's sum was only possible if it was chosen when the player hit).
- hit_or_stand_threshold(): This function determines the player's suggested move, if possible, given their current probability tree state, their target sum, and threshold. The subtrees of the root's node compute the sum of cards remaining that would cause it to bust. That sum divided by the total cards in that deck produces the proportion in which the player should stand, and if that percentage is less than or equal to the threshold, they will suggest hitting, as the odds of busting with respect to their limit is not over their threshold.
- get_subtrees(): Return a mapping of card types to a tuple of their representative Probability tree object and the remaining cards that they could've chosen from their game state's deck. The returned dictionary is in a similar format to the deck's dictionary.
- find_subtree_by_sum(): Return a probability tree based on their root's current sum value and their card type. This probability tree that we want to return must be a valid sum that was previously computed from its parent node's subtrees. This method is important in following down to a specific Probability Tree node so that when the player hits and that card is drawn, the probability was accounted for and can have its threshold recalculated to suggest the next move.

Our stats.py file handles a large portion of the visual aspects of code. It has 3 main groups of functions: graphing functions, data generation functions, and back-end functions that the GUI utilizes to display statistics. Our two main libraries for this file are pandas and plotly express, which we use in more depth to manipulate and extract data for graphing purposes.

We also created a graph based on our probability tree using the NetworkX and Matplotlib libraries. This was done using two functions, aptly named tree_to_graph to create a graph using a given ProbabilityTree, and draw_graph to then create a visual representation of this graph. tree_to_graph is implemented using recursion, going down each subtree of the Probability Tree and linking them to the larger graph. It uses NetworkX's dictionary constructor,

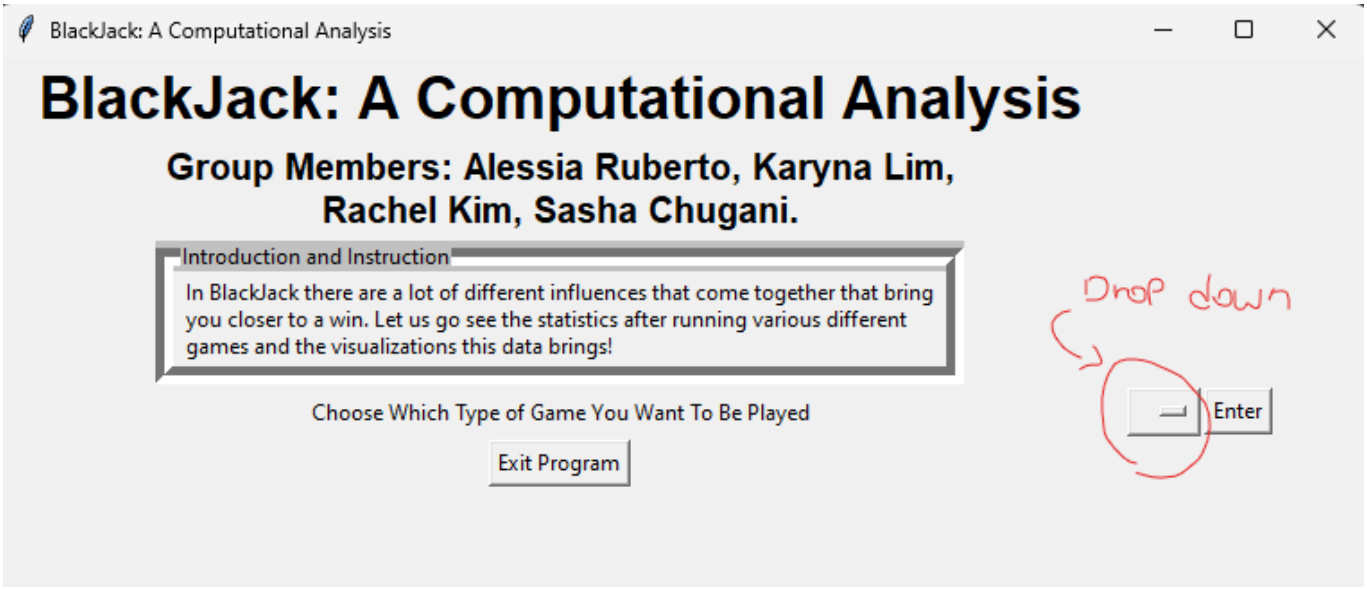
where we create a dictionary to map nodes to one another, and then create a graph from this information. Each node in this graph is a tuple, with the first element of the tuple being the numerical value of the sum, while the second acts as a unique identifier. This is done so that even when there are duplicates of the sum value which there are many due to the nature of the game- each node connects to its proper root. Each of the nodes and edges in the corresponding graph can be seen and checked by calling list('graph-name'.nodes) and list('graph-name'.edges), respectively. However, with larger graphs, it can be both difficult and tedious to try and follow these lists, which is why we also created our draw_graph function. This function also uses NetworkX, as well as Matplotlib's pyplot, and takes the inputted graph to create a visual drawing of the graph, so it can be more easily seen and understood instead of trying to look at all the nodes and edges in a list.

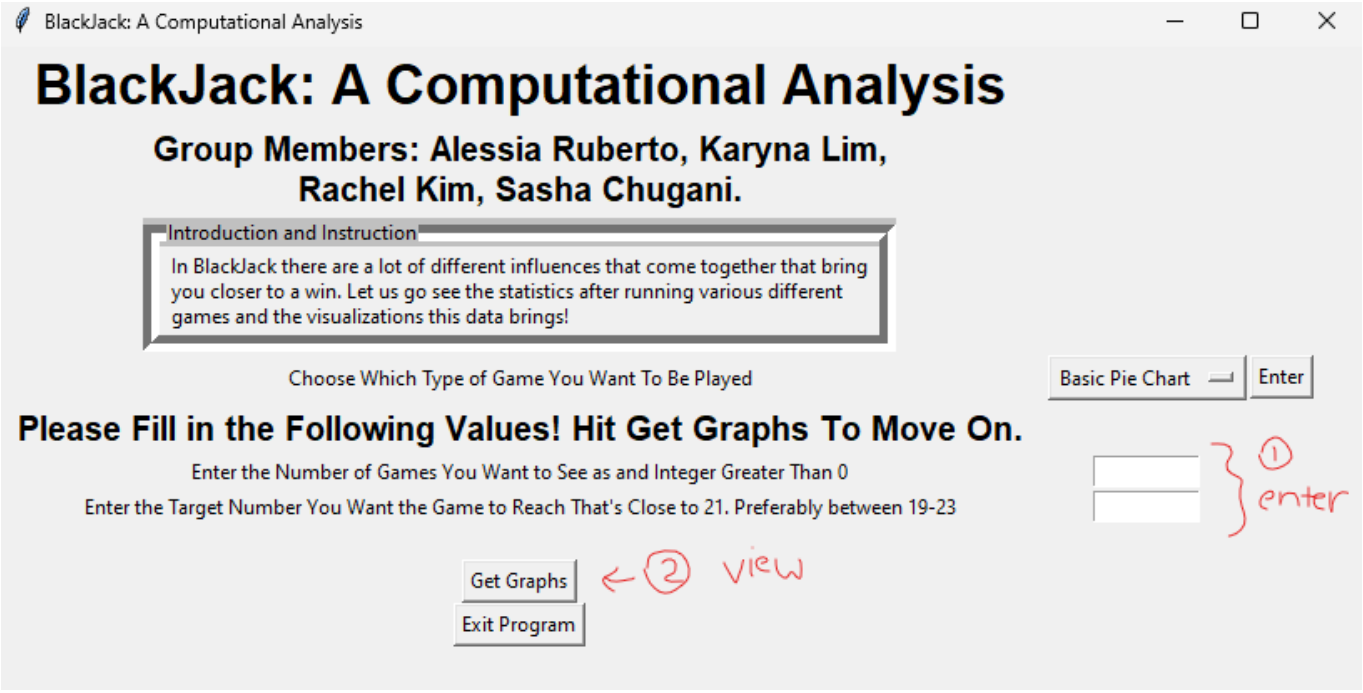
Our gui.py launches an interactive page in which other people can input a target value, a number of games, and if needed a threshold value to get a graph in return of the win loss ratio using their choice of strategy. The file first creates a function that launches the whole program when you run it in the console. In the function, the first thing it does is create a widget titled BlackJack: A Computational Analysis, that displays the title and group members which both were created using Label. It then adds a frame that introduces the point of the interactive user interface which uses LabelFrame and Message to create a frame around the text of the message. It then creates a drop-down menu using OptionMenu where the person using the GUI can choose which type of visualization they want to see being done with options that were placed in the function which included different ways to play the game and pre-made dataset visualizations. The options for choosing the game include Basic Pie Chart, Strategical Tree Pie Chart, and Dual Comparison Bar Graph. The premade datasets are ones that the user of the interactive module can choose to see which will display either visualizations of the data, a tree structure, or a graph structure. After a selection is made in the drop-down menu, we use Button to create an enter button then once something has been selected, the command of the button takes it to a function labeled get_necessary_inputs. The function then uses a series of if statements to see what the selection was and depending on that it uses helper functions to determine the input that should appear using a Label and an Entry. This step opens entry forms where numbers can be placed where the same function collects these numbers and uses them to create the graph the user wants calling the file stats.py.

Instructions

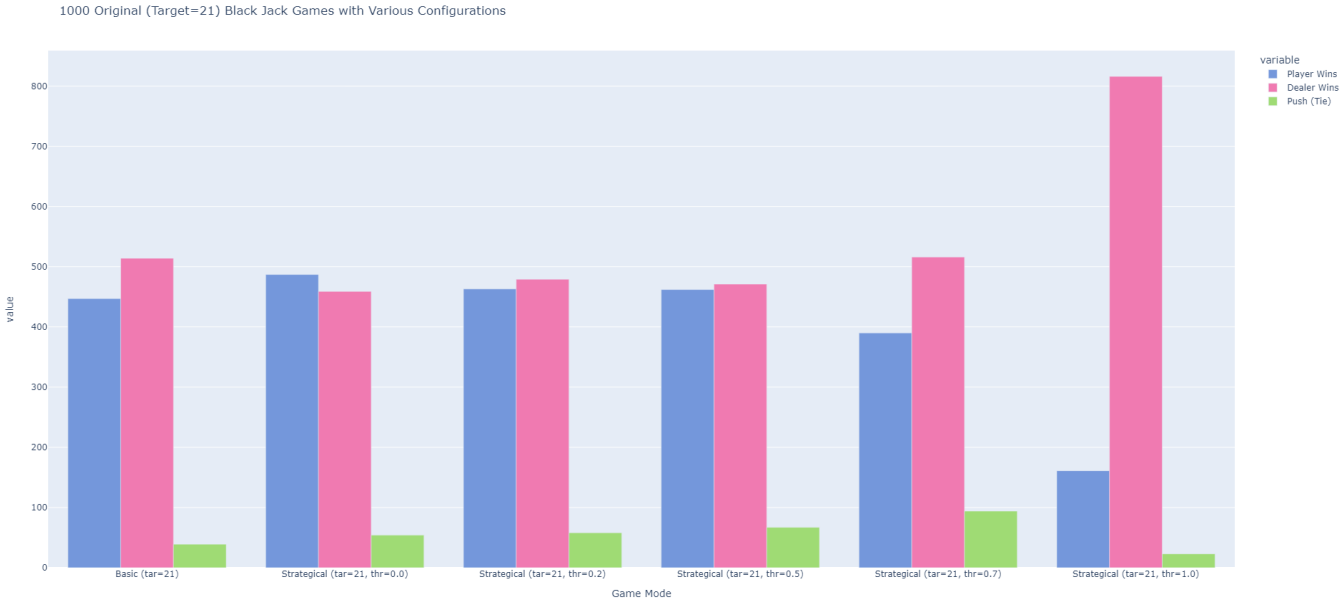
Firstly, all of our python modules should be placed in the same directory as "main.py." All of our data sets and example outputs are located in the zip file named "data_and_outputs" which has been uploaded to MarkUs The folders in this zip file should be placed in the same directory as the python modules (i.e. leave the files in the folders). For example, it should be placed such that the correct location to call a .csv file in data is 'data/some_file_name.csv.' It is necessary that these are placed correctly for certain graphing options to work.

Once everything is in the correct location, simply open and run "main.py." A GUI should pop up which looks like the following:





Once you have made a selection for what you would like to view, the output should launch in a browser. (Note: our default is Chrome, though we expect it will work with other browsers). We also have pre-generated graphs images in the zip file for reference. Here is an example graph output after selecting "Large Generated Example Data Original."



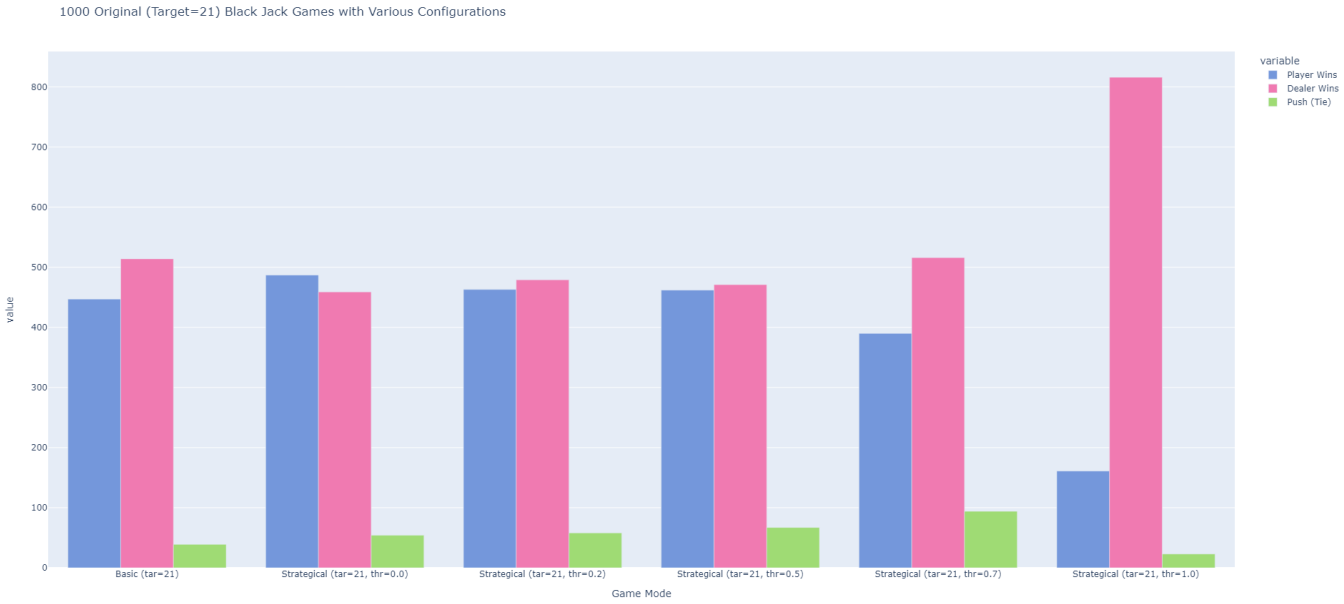
Changes

Our project plan remained steady between the proposal and final submission with most steps staying relatively similar, but there were a few slight changes we made. In our original plan we stated that if there was time we would try to "display the probability tree, and possibly a GUI of the game being played, so that users can visually understand how the algorithm for the player, for example, obtains its cards from the deck given the game move probabilities . . . would allow users to interact with our algorithm by submitting values for the certain parameters, such as the target number for the game's round and other rules or concepts if possible" As we approached this, we listened to input from the TA and used tkinter to set up an interactive interface. This interface wasn't a demonstration of the game

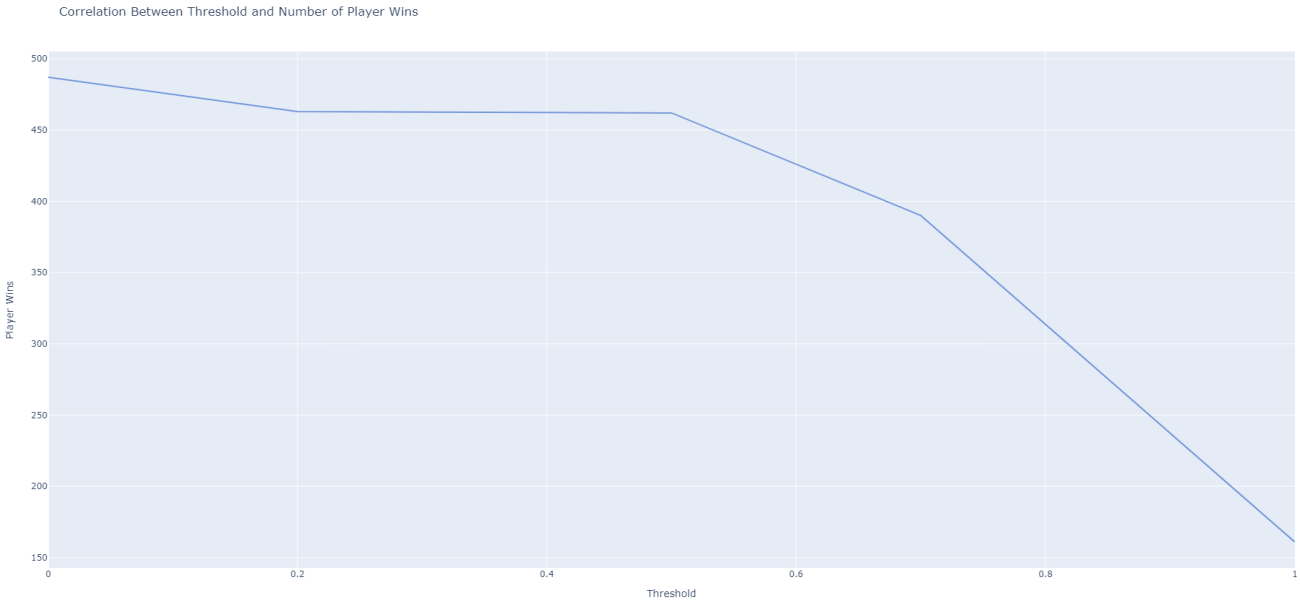
being played, but it does showcase a singular tree of potential values as an example of what the tree function looked like as well as a set graph of these values. This interface also allows a user to choose how many games they want to run, a target value, and a threshold value depending on the game they want to see the statistics of. After a user inputs these values, the interface will return a graph of wins, losses, and ties using the information inputted. This interface also has a complete and fixed large dataset which a user can select and see the graphs of its wins, losses, and ties.

Discussion

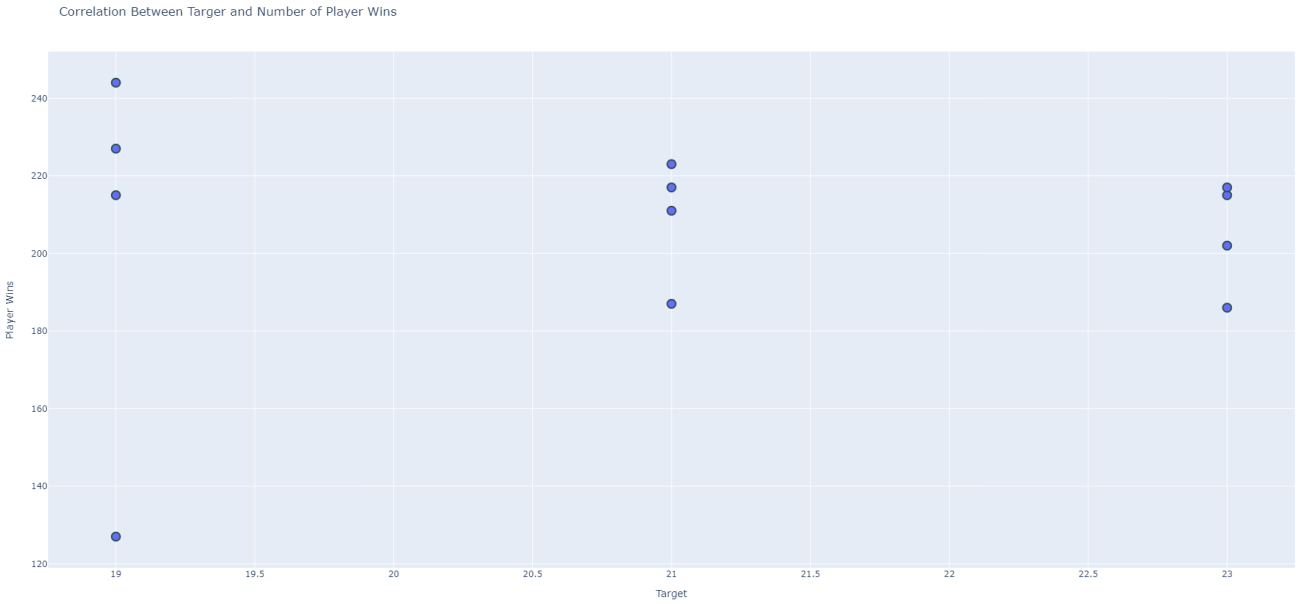
We were successfully able to analyze the results of our computations to discover the most optimal way to play Black Jack. According to our data, the best-performing strategy for the original version of the game is following our probability tree strategy at a threshold value of 0.0. However, it only performs slightly better than the other options, hovering at around a 54% win rate as opposed to 45% when played using the basic strategy. This result was expected because as stated previously, Black Jack is heavily randomized. In some circumstances, no matter the strategy, you will be prone to a loss. However, overall it successfully brings the player to an advantage.



With our probability tree strategy, again, we noticed the best-performing threshold value was 0.0. After testing several different threshold values, we noticed a negative correlation between the threshold value and the win rate. The following graph shows a clear demonstration of the trend:



Extending to the experimental phase of our project, we modified the game rules to see if the target number has an effect on the player's wins. We tested the game at targets near 21, such as 19 and 23. Overall, we found a negative correlation between the target number and player wins.



In terms of obstacles, by far our largest was our inexperience with utilizing the libraries. With pandas, although we knew how to manipulate data using a list representation of a table, we were unfamiliar with how to manipulate the same format of data with pandas. This led to a lot of frustration with doing simple data manipulations because we were unfamiliar with the library's capabilities and limitations. Ultimately, with research and patience, we were able to make it work.

Similarly, we had faced setbacks creating our own graphs with plotly for the same reasons. However, we ultimately were able to learn how to create many types of graphs and customize certain aspects like colours, size, axes, etc.

We also underwent various obstacles when understanding tkinter and the functions that came with it. There were times when we weren't sure how to develop the interactive side and connect it with other functions. For example, there was an issue with us being unable to reset functions that we had to learn how to work around to follow our initial plan.

Another new library we used and experienced obstacles with was NetworkX. Initially, it didn't seem like it would be too challenging, however as we started coding, we found ourselves tripping up several times, sometimes even for the smallest reasons, because we were again, unfamiliar with the library and its functions. Like with the other libraries though, in the end, we were able to use it successfully, learning to create graphs with new, interesting constructors and methods. Plotting the graph visually posed more challenges, especially with the positioning of the nodes, because we had to come up with our own specific positions as none of the preexisting layouts offered by NetworkX were suitable for the type of graph we wanted to present. This was made even more difficult by the sheer number of nodes that could be produced depending on the Probability Tree. Still, we persevered again and did our best to create the most appropriate visual we were able.

There are several potential updates we can explore in the future, such as changing our draw_card function to account for varying targets. We could also create an actual interactive pygame feature using a GUI, as all our cards are properly labeled and organized. This would mean a cleaner, more easily understood visualization of the game, which can be made even further interactive by introducing our probability methods as 'hints' of sorts for the user if they are unsure of which move to make. Another potential update we can explore is creating an option to play a whole game with the same deck. If we were to do this, we would have to add an attribute keeping track of the status of the deck, and methods signaling when it is empty, thus meaning the game is over. This would then also mean a minor tweak to our current Blackjack Game implementation that re-initializes the game's round statuses.

References

"10 Minutes to Pandas." 10 Minutes to Pandas - Pandas 1.5.3 Documentation, https://pandas.pydata.org/docs/user_guide/10min.html.

A Newbie Guide to Pygame - Pygame V2.1.4 Documentation, https://www.pygame.org/docs/tut/newbieguide.html.

Blackjack, https://bicyclecards.com/how-to-play/blackjack/.

Blackjack Rules for Dealers: How Does the Dealer Affect the Game? https://bitcasino.io/blog/live-dealer/important-blackjack-dealer-rules-to-remember.

"Blackjack Strategy Charts - How to Play Perfect Blackjack." Blackjack Apprenticeship, 2 June 2022, https://www.blackjackapprenticeship.com/blackjack-strategy-charts/.

"Box." Box Plots in Python, https://plotly.com/python/box-plots/.

"Codemy.com." YouTube, YouTube, https://www.youtube.com/@Codemycom.

"How to Get the Value of an Entry Widget in Tkinter." Tutorials Point, https://www.tutorialspoint.com/how-to-get-the-value-of-an-entry-widget-in-tkinter: :text=Let%20us%20suppose%20that%20we,or%20display%20the%20entered%20value.

Fitzpatrick, Martin. "Tkinter Layouts, Designing Python Gui." Python GUIs, Python GUIs, 13 Mar. 2023, https://www.pythonguis.com/tutorials/use-tkinter-to-design-gui-layout/.

"Gallery." Gallery - NetworkX 3.0 Documentation, https://networkx.org/documentation/stable/auto_examples/index.html.

"How to Change Border Color in Tkinter Widget?" GeeksforGeeks, 23 Nov. 2021, https://www.geeks forgeeks.org/how-to-change-border-color-in-tkinter-widget/.

"Plotly." Plotly Python Graphing Library, https://plotly.com/python/.

"Python Gui - Tkinter." GeeksforGeeks, GeeksforGeeks, 4 Jan. 2023, https://www .geeksforgeeks.org/python-gui-tkinter/.

"Tutorial — NetworkX 2.5 Documentation." Networkx.org, networkx.org/documentation/stable/tutorial.html.

"Line." Line Charts in Python, https://plotly.com/python/line-charts/.

"Draw_networkx — NetworkX 3.0 Documentation." Networkx.org, https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw_networkx.html