Gambler's Triumph

You've just received your first bonus—and you're ready to risk it all on a game of blackjack at the local casino. But the rules here aren't your standard Vegas fare, so pay close attention. Let's break it down:

Gameplay Rules

1. Number cards (2–10) have a value equal to their number (e.g., a 7 is worth 7 points).
2. Face cards (King, Queen, Jack) are worth 10 points each.
3. Aces always count as **11** (unlike regular blackjack, where they can be 1 or 11).

You and the dealer each have your own "deck." In reality, each deck is an infinite stream of cards generated by a specific but unknown distribution. **In every game, first your cards are dealt and the dealer's.** Here's how the game unfolds: the dealer deals cards to you one by one. At each step, you choose between:

1. `Hit`: Take another card.
2. `Stand`: Stop drawing cards.

Your goal is to maximize your total without exceeding 21. If you go over, you `bust` and lose immediately. For example, if your first card is a Queen (10 points) and you hit, drawing a 9, you now have 19. Standing at 19 is usually wise—another hit comes with a high risk of busting. Once you `stand` (without busting), it's the dealer's turn. They unfold their cards one by one. Their strategy is fixed:

1. The dealer hits while their total is 16 or less.
2. As soon as their total exceeds 16, they `stand`.

Additionally, if after any draw the dealer's total is 16 or below, you can choose to `surrender`, forfeiting half your bet and keeping the rest. You may also surrender right after you `stand`—as long as you haven't busted—before the dealer draws their first card. Suppose you've bet $1. These are the possible outcomes:

1. If you `bust`, you lose your $1 immediately, ending up with $0.
2. If the dealer `busts`, you win $1 (leaving you with $2 in hand).
3. If the dealer's total is 16 or below, you may `surrender`, keeping $0.50.
4. If neither busts, the higher total wins. A win leaves you with $2 in hand; a loss leaves you with $0.
5. If both totals match, it's a tie (a "push") and you keep your $1.

You walk into the casino with cash to spare. `Assume each of your bets is small and consistent, so you never risk going fully broke.`

The Winning Edge: Decoding the Deck's Secret

Normally, the odds are against you. But thanks to your bonus, you've acquired a special spyglass that reveals how the casino really works.

Deep in the casino's backend, the decks aren't shuffled—they're generated via an intricate autoregressive process. Each new card value is based on previous ones, with some Gaussian noise to keep things spicy. Your spyglass reveals the real-numbered "spy" values behind the curtain—giving you a rare glimpse into what drives the game.

The casino creates each infinite card stream using an autoregressive process on these spy values, which are then mapped to a card number. Each table has two independent streams: one for the dealer, one for the player (you). Figure 1 shows an example rollout of such a series.

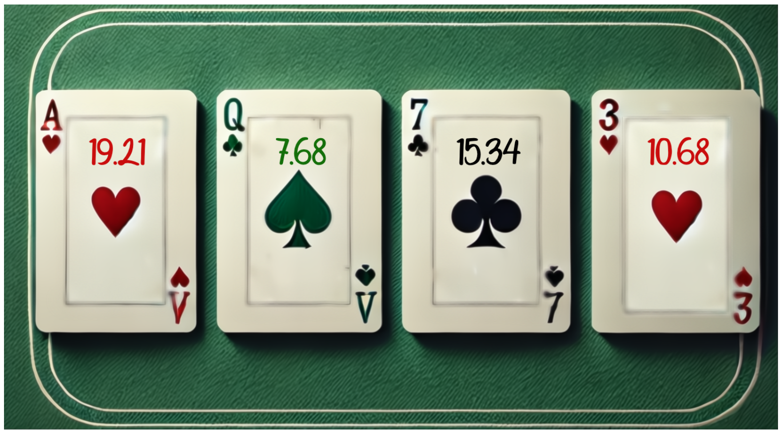

Figure 1: A sample run of 4 draws. The series on top of the cards is the spy series only visible to you.

Problem Setup

We've provided data from 5 tables. For each table, there are two time series: one for the dealer, one for the player. Each series is a $T \times 2$ matrix, where T is the number of observations. Each row includes:

1. The spy value (real-numbered output from the autoregressive process)
2. The observed card value (with Aces = 11, and K/Q/J = 10)

We recommend splitting each dataset into training and validation sets to avoid overfitting and ensure generalizability.

We advise you to use the defined `data_loader` (and `sicily_data_loader` in part 6) function to load the data for a stream. It takes `path of the data`, `table_index`, `type of stream` (”player” or ”dealer”) and returns a numpy array of dimension $N \times 2$ corresponding to N draws. Each draw is then (spy value, card value).

`Please read through the entire problem before starting.`

Guidelines

Modular Approach

Each part builds on the last, but you're free to approach them independently. If you get stuck on one, feel free to shift focus and return later.

Parts 1 & 2 – Building the Foundation

These parts lay the groundwork for your strategy. While it might be tempting to jump in with heavy-duty stats, well-placed, sharp insights might be faster and more effective.

Part 3 – Forecasting the Series (Independent)

This section stands alone. Here, you'll model the spy series for each deck predicting future values based solely on historical data. Bringing out the heavy artillery with some nuanced target-setting or.. living and dying by the phrase "first-principles are the final principles" – choose your battles. Don't worry if Parts 1 and 2 aren't fully polished—this part is independent and can be approached on its own merits.

Part 4 – Blackjack Strategy (Depends on Parts 1, 2 & 3)

Bring together your insights and models to craft a winning game strategy. Your decision-making should be directly informed by earlier parts.

Beyond – Parts 5 & 6

Once you've got a handle on the fundamentals, expand your strategy to handle multi-game scenarios (Part 5) and cross-table plays (Part 6). Think creatively, build on your previous insights, and adapt as needed.

Remember, creativity and flexibility are your best tools. Use each part's insights to build a comprehensive strategy, and don't hesitate to pivot if one approach hits a roadblock. Happy strategizing!

Solution Setup

We recommend solving each part locally on your own machines and uploading the final code for each part in the desired format. This approach will allow for faster iterations and greater flexibility. Additionally, as you will be required to reuse your models and prediction functions in later sections, it is advisable to save them locally to ensure ease of access and consistency throughout the subsequent tasks.

Along with submitting code for each part individually, a strong submission should include a well-documented Jupyter notebook (or equivalent in your preferred language) that clearly demonstrates your thought process. You need to submit it in Question 1. Your code should be clean, readable, and thoroughly commented. Include relevant visuals, plots, and well-presented data to support your analysis and conclusions. Since we will not be running your notebook, ensure that all outputs are visible and that the notebook is neatly formatted and self-contained. This notebook is the most important part of your submission and will carry the greatest weight in the evaluation, so please take the time to present your work clearly and thoughtfully.

1 Dealer's Doom

To identify profitable tables, estimate the empirical apriori probability that the dealer busts at each table (ignore player actions like `surrender`). For this, simulate the dealer's runs using the data given for each table and compute this probability. Assume that one epoch over the given streams is sufficient for a good approximation of this probability. Complete the `bust_probability` function in `calc_probability.py`. Validate your code using `analysis.ipynb`. Propose at least two additional metrics that capture your "willingness to play" at a table. Explain the strengths and weaknesses of each in the notebook you will submit.

2 Sherlocking the Cards

Next, deduce the relationship between a spy-series value and the corresponding card value. You need to define a single function (which is deterministic and has zero error) that translates the spy value into the card's blackjack value. This function applies universally (across all tables and for both dealer and player streams). Implement `get_card_value_from_spy_value` so it returns the correct card value (2–10, or 10 for face cards, 11 for Aces) when given the spy value.

3 Spy Series Seer

If you were unable to complete the previous section, there is no need for concern, as this part is independent of that. Your task is to forecast future spy values. For each of the 5 tables (dealer and player), build a model to predict the next spy value in the series based on its previous values. You have 10 total series to model. You're free to use any architecture you like—linear regression, neural networks, decision trees, etc. and do any feature engineering you find helpful. Your are only allowed to change `Player.py`. Additionally, we have

provided a `score.py` script to evaluate your predictions. This script prints the Mean Squared Error (MSE) for both the dealer and the player on a specified table. You can modify the evaluation table by changing the `table_index` variable. It can take values in the set $\{0, 1, 2, 3, 4\}$. In the end, you'll submit 10 trained model objects, one for each of the 10 series. Each model will be evaluated against a hidden test set.
As illustrated in `score.py`, the evaluation process begins by initializing your player class with the desired `table_index`. Use the `__init__` method in your implementation to load all required models and perform any necessary initializations. Once the initialization is complete, the functions `get_player_spy_prediction` and `get_dealer_spy_prediction` should be callable multiple times in any order without issue. Please note that some series may not be inherently suitable for modeling; in such cases, it is advisable to make a reasonable judgment and proceed to the subsequent parts without spending excessive time on less promising series.
In the notebook that you submit in Part 1, please describe:

- Which metric you optimized for modeling each series.
- Which model architecture you chose and why.
- Any feature engineering steps you took and why.

After completing your implementation, ensure that running the following command executes without any errors : `python3 score.py`. Successful execution of this script without crashes indicates that your code is compatible with our hidden evaluation framework, and it will run correctly during final testing as well.

4 One Shot Showdown

With your forecasting model (part 3) and card-value function in hand (part 2), it's time to play. You get one shot at each table. During your single game at each table, you can see all the cards that have been dealt so far (both to you and the dealer) along with their spy values.

Complete the `get_player_action` method in `MyPlayer` class. It should always return a string, depending on whose turn it is:

- Player's turn: `stand` or `hit`
- Dealer's turn: `surrender` or `continue`

Your goal is to maximize your expected payoff for that single game. You are supposed to fill the file `Player.py`. Use the provided `score.py` script with your chosen policy to see how it performs on the validation set. It splits the dataset into runs of 20 cards each and treats each split individually. It then simulates a blackjack game on each split and prints the overall payout. **The current config runs the simulation script on the entire train dataset and the results might be susceptible to overfitting. It is advised to change it to a validation set and validate your strategy on that**. Some tables might not be profitable, so don't lose sleep over those edge cases.

After completing your implementation, ensure that running the following command executes without any errors : `python3 score.py`. Successful execution of this script without crashes indicates that your code is compatible with our hidden evaluation framework, and it will run correctly during final testing as well. A `debug` option is also provided, which prints detailed logs for each game. This allows you to review and analyze the sequence of actions taken during gameplay.

5 Marathon of Twenty-One

Now you want to develop a strategy to play multiple games at each table. A straightforward approach is to reuse your single-game strategy for every hand. Alternatively, you can try to refine it for a globally optimal strategy across multiple hands. Consider whether a fully exhaustive approach is computationally feasible, or if you'll need a good approximation. In any case, you'll use the same approach for all tables. Again, not every table may yield a profit, so don't get bogged down by the tough ones. You are supposed to fill the file `Player.py`. Use the provided `score.py` script with your chosen policy to see how it performs on the validation set. It simulates 200 games of blackjack. **The current config runs the simulation script on the entire train dataset and the results might be susceptible to overfitting. It is advised to change it to a validation set and validate your strategy on that**. Some tables might not be profitable, so don't lose sleep over those edge cases.

After completing your implementation, ensure that running the following command executes without any errors : `python3 score.py`. Successful execution of this script without crashes indicates that your code is compatible with our hidden evaluation framework, and it will run correctly during final testing as well. A `debug` option is also provided, which prints detailed logs for each game. This allows you to review and analyze the sequence of actions taken during gameplay.

6 Sicilian Synergy

Word of your blackjack skills has reached the top brass. You're invited to a high-roller casino in Sicily where the card streams follow a similar pattern to your hometown—but they might be slightly out of sync, that is some series might just be shifted versions of some other series. You're given a challenging new table to play on. However, you're also allowed to simultaneously play two other tables from your hometown. So now at any point in time, you have 3 decisions to make, one for each table. Also, the dealer(player) run at any table starts only after the players(dealers) on all tables have completed their rollout. So, when the player run ends on all tables, the dealer starts opening her cards for each table. Only after dealers turn ends at all three tables does the whole round end. Please look at the function signature of `get_player_action_multi` for more clarity. If certain streams are correlated, you might be able to exploit those connections to boost your winnings at the Sicilian table. Assuming you now are able to play at these three tables simultaneously, can you make a coordinated strategy to exploit these information dissemination lags which is profitable?

As in the previous parts, you are required to complete the `MyPlayerMulti` class in `PlayerMulti.py`. Within the `choose_tables` function, you must specify the indices of any two of the previous tables you wish to play on in the form of a list. To the end of this list, we'll append the sicilian table. The `get_player_action_multi` should return a list of actions for each table (in the order specified above) in each run. To validate your implementation, use the provided `score_multi.py` script. Ensure that executing the following command completes without any errors: `python3 score_multi.py`. Successful execution confirms that your code is functioning as expected and is compatible with the evaluation framework. Good luck—and may the cards (and spy values) be ever in your favor!