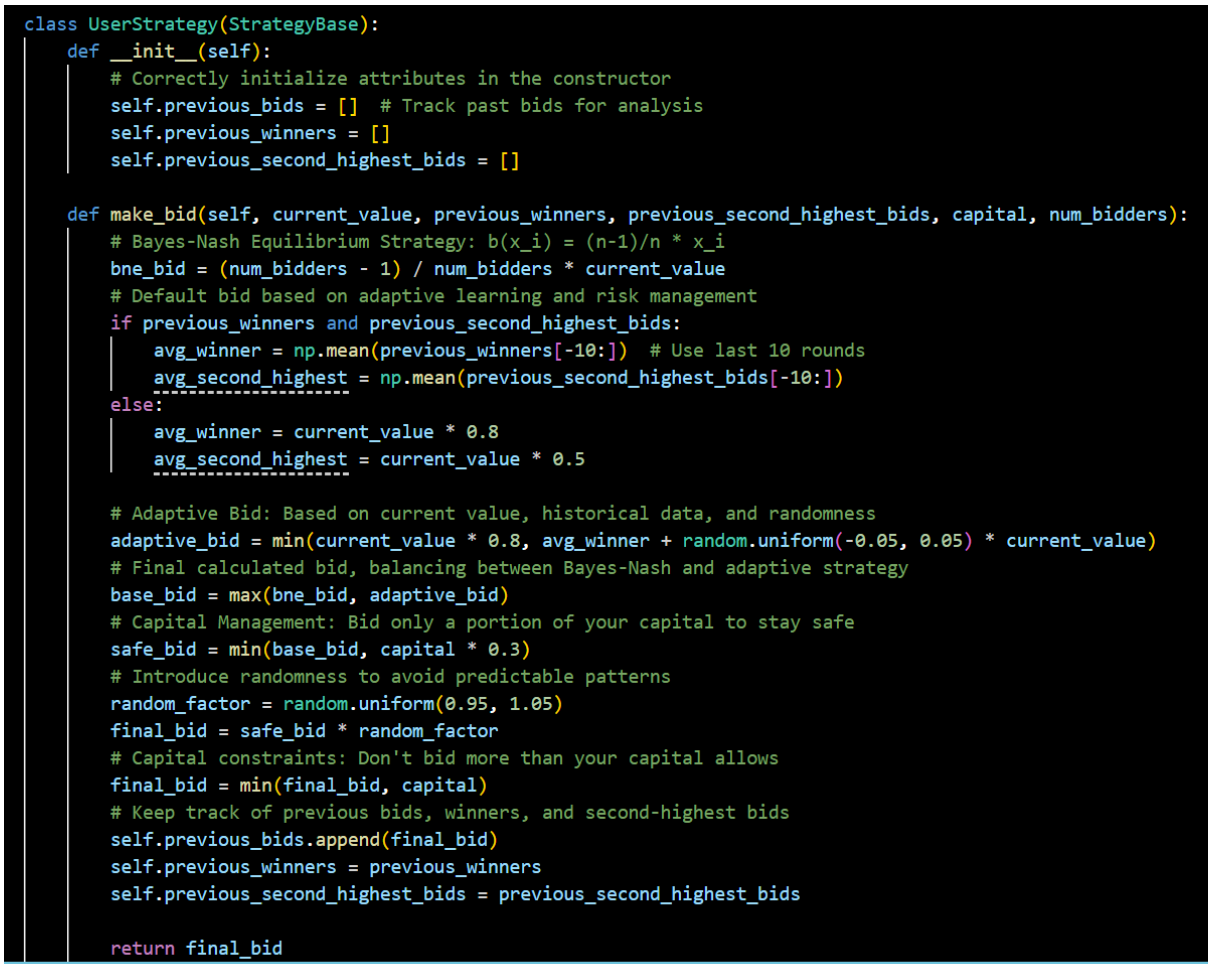
REPORT OF 3 STRATEGIES

Mohini Katyal, EE22B122

Variation 1:



```python
class UserStrategy(StrategyBase):
    def __init__(self):
        # Correctly initialize attributes in the constructor
        self.previous_bids = []  # Track past bids for analysis
        self.previous_winners = []
        self.previous_second_highest_bids = []

    def make_bid(self, current_value, previous_winners, previous_second_highest_bids, capital, num_bidders):
        # Bayes-Nash Equilibrium Strategy: b(x_i) = (n-1)/n * x_i
        bne_bid = (num_bidders - 1) / num_bidders * current_value
        # Default bid based on adaptive learning and risk management
        if previous_winners and previous_second_highest_bids:
            avg_winner = np.mean(previous_winners[-10:])  # Use last 10 rounds
            avg_second_highest = np.mean(previous_second_highest_bids[-10:])
        else:
            avg_winner = current_value * 0.8
            avg_second_highest = current_value * 0.5

        # Adaptive Bid: Based on current value, historical data, and randomness
        adaptive_bid = min(current_value * 0.8, avg_winner + random.uniform(-0.05, 0.05) * current_value)
        # Final calculated bid, balancing between Bayes-Nash and adaptive strategy
        base_bid = max(bne_bid, adaptive_bid)
        # Capital Management: Bid only a portion of your capital to stay safe
        safe_bid = min(base_bid, capital * 0.3)
        # Introduce randomness to avoid predictable patterns
        random_factor = random.uniform(0.95, 1.05)
        final_bid = safe_bid * random_factor
        # Capital constraints: Don't bid more than your capital allows
        final_bid = min(final_bid, capital)
        # Keep track of previous bids, winners, and second-highest bids
        self.previous_bids.append(final_bid)
        self.previous_winners = previous_winners
        self.previous_second_highest_bids = previous_second_highest_bids

        return final_bid
```

Strategy Logic:

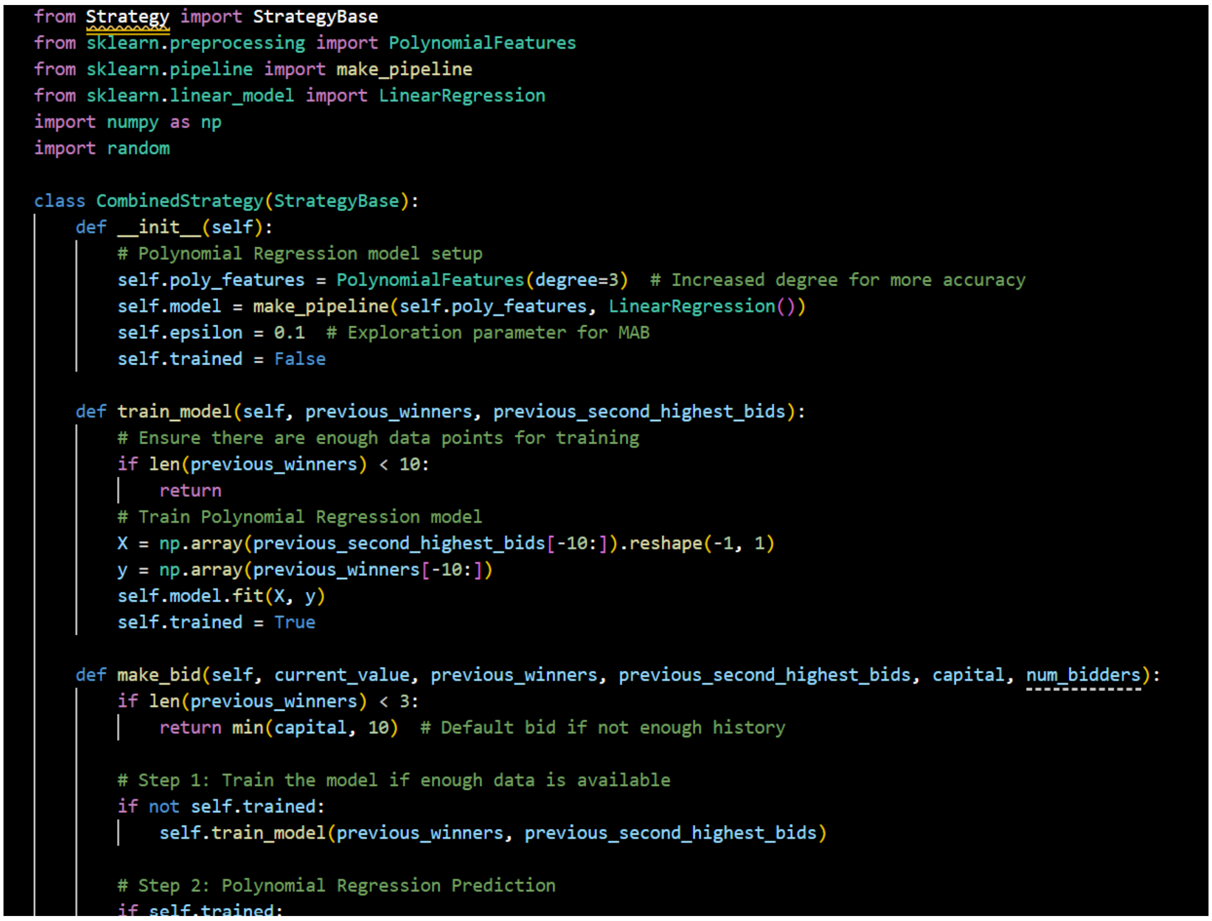
- **Bayes-Nash Equilibrium Strategy (`bne_bid`): This follows the Bayes-Nash Equilibrium formula, which is a theoretical optimal bidding strategy in many auction formats.**
- **Adaptive Learning:**
    - **The strategy uses historical data (previous winners and second-highest bids) to adjust the bid.**
    - **If historical data is available, it calculates the average of the last 10 rounds for both winners and second-highest bids. Otherwise, it estimates values based on the current round's value.**
- **Adaptive Bid:**
    - **An adaptive bid is calculated by combining historical data, randomness, and risk management (setting a percentage of the current value as the bid).**
- **Capital Management:**
    - **The bid is capped at a portion of the available capital (25% here) to ensure the player doesn't run out of capital too quickly.**
- **Randomness:**

  - **A random factor (between 0.97 and 1.03) is introduced to the bid to make the strategy less predictable.**
- **Final Bid:**
  - **The final bid is the minimum between the calculated bid (`base_bid`) and the available capital. The goal is to avoid overbidding.**
- **Tracking History:**
  - **The method also updates the history of previous bids, winners, and second-highest bids for use in future rounds.**

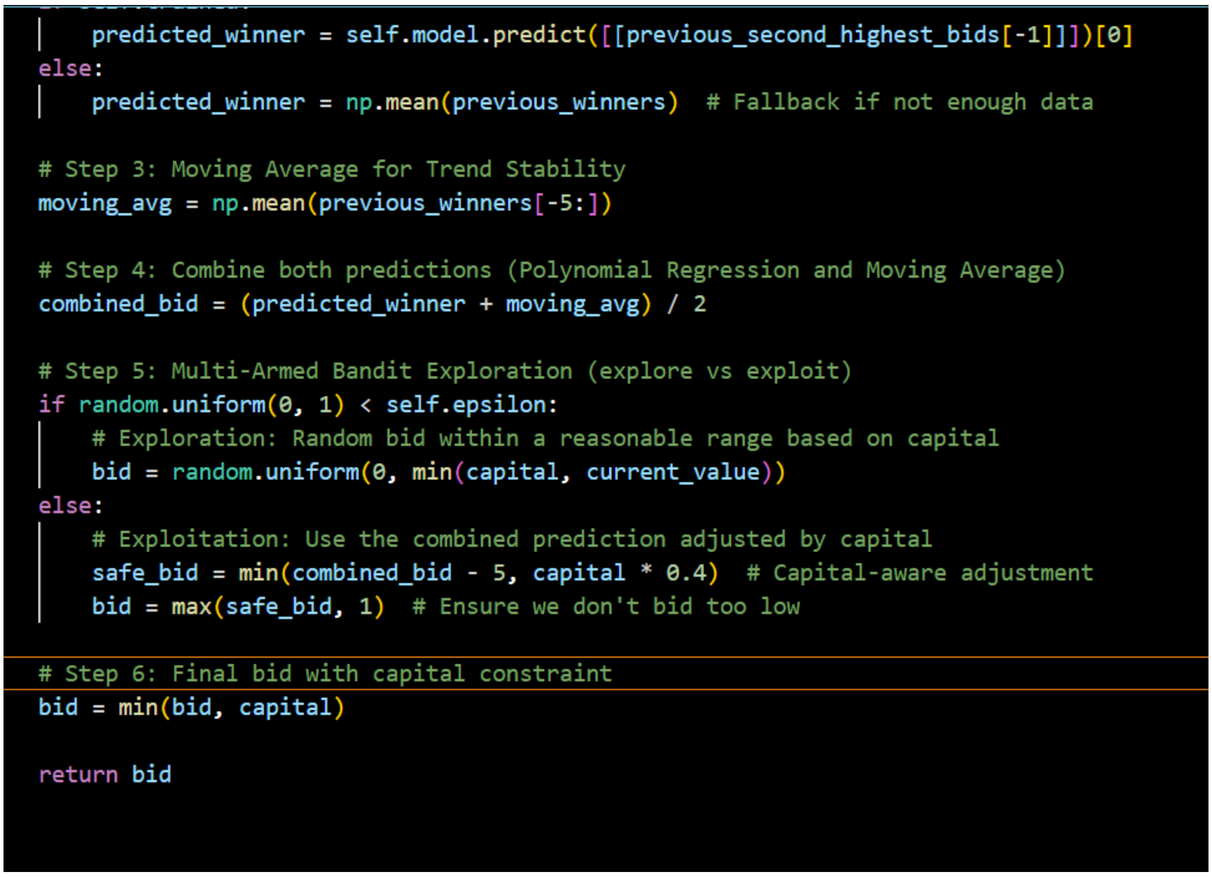
In summary, this strategy uses a combination of game theory (Bayes-Nash Equilibrium), adaptive learning based on previous rounds, capital management, and randomness to optimise the bidding process.

The approach is designed to balance risk and reward, ensuring that the player doesn't deplete their capital too quickly while staying competitive in the auction.

Variation 2:



```python
from Strategy import StrategyBase
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LinearRegression
import numpy as np
import random

class CombinedStrategy(StrategyBase):
    def __init__(self):
        # Polynomial Regression model setup
        self.poly_features = PolynomialFeatures(degree=3)  # Increased degree for more accuracy
        self.model = make_pipeline(self.poly_features, LinearRegression())
        self.epsilon = 0.1  # Exploration parameter for MAB
        self.trained = False

    def train_model(self, previous_winners, previous_second_highest_bids):
        # Ensure there are enough data points for training
        if len(previous_winners) < 10:
            return
        # Train Polynomial Regression model
        X = np.array(previous_second_highest_bids[-10:]).reshape(-1, 1)
        y = np.array(previous_winners[-10:])
        self.model.fit(X, y)
        self.trained = True

    def make_bid(self, current_value, previous_winners, previous_second_highest_bids, capital, num_bidders):
        if len(previous_winners) < 3:
            return min(capital, 10)  # Default bid if not enough history

        # Step 1: Train the model if enough data is available
        if not self.trained:
            self.train_model(previous_winners, previous_second_highest_bids)

        # Step 2: Polynomial Regression Prediction
        if self.trained:
```


```
    predicted_winner = self.model.predict([[previous_second_highest_bids[-1]]])[0]
else:
    predicted_winner = np.mean(previous_winners)  # Fallback if not enough data

# Step 3: Moving Average for Trend Stability
moving_avg = np.mean(previous_winners[-5:])

# Step 4: Combine both predictions (Polynomial Regression and Moving Average)
combined_bid = (predicted_winner + moving_avg) / 2

# Step 5: Multi-Armed Bandit Exploration (explore vs exploit)
if random.uniform(0, 1) < self.epsilon:
    # Exploration: Random bid within a reasonable range based on capital
    bid = random.uniform(0, min(capital, current_value))
else:
    # Exploitation: Use the combined prediction adjusted by capital
    safe_bid = min(combined_bid - 5, capital * 0.4)  # Capital-aware adjustment
    bid = max(safe_bid, 1)  # Ensure we don't bid too low

# Step 6: Final bid with capital constraint
bid = min(bid, capital)

return bid
```

1. Polynomial Regression for Predicting Future Winning Bids

- **The core idea is to use past data (specifically, the highest and second-highest bids from previous rounds) to predict the winning bid in the next round.**
- **Polynomial regression is employed here, which fits a curve to the data instead of a simple straight line. This allows the model to capture non-linear patterns, which might be present in how bids evolve over time.**
- **The bot trains this model on the last 10 rounds of data (if available) to predict what the winning bid will likely be in the current round.**

2. Moving Average to Capture Recent Trends

- **The bot also calculates a moving average of the winning bids from the last 5 rounds to get a sense of the short-term trend.**

3. Combining Predictions for Stability

- **The strategy doesn't rely on just one prediction method. It combines the predictions from the polynomial regression model with the moving average to arrive at a more robust estimate of what the winning bid might be.**

4. Multi-Armed Bandit Approach for Exploration and Exploitation

- **The bot occasionally "explores" by making a random bid, even if it's not based on the prediction model.**
- **This exploration happens with a probability of 10% (`epsilon = 0.1`), meaning that in 1 out of 10 rounds, the bot will place a random bid within a reasonable range (based on its capital and the value of the current round).**

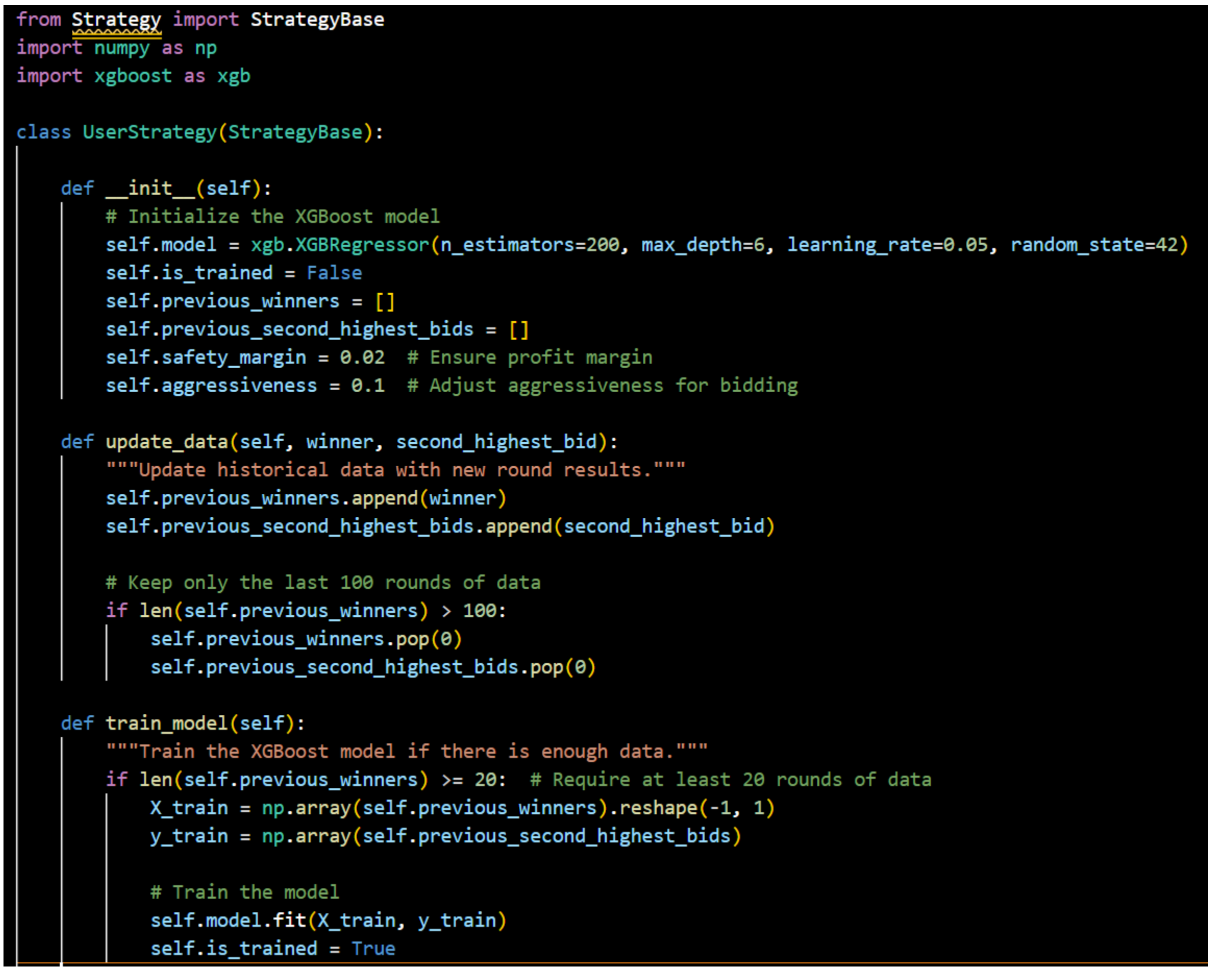
- For the remaining 90% of the rounds, the bot will **exploit** its model and bid based on the predicted winning value.

5. Capital-Aware Bidding

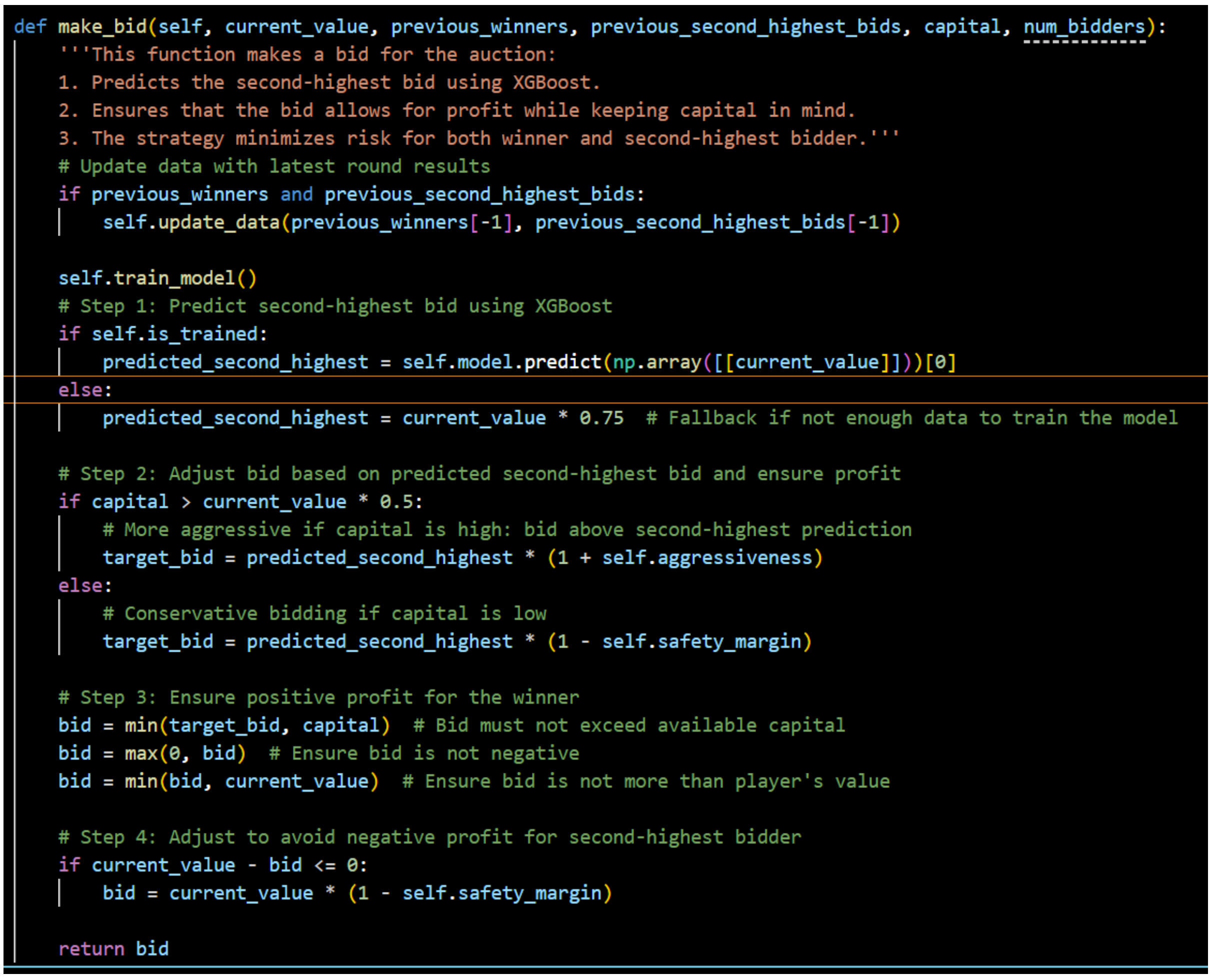
- The bot carefully adjusts its bid based on how much capital it has left. Even if the predicted winning bid is high, the bot won't risk all of its money in a single round.
- The strategy includes a capital constraint where the bid is capped at 40% of the bot's remaining capital. This helps prevent the bot from running out of money too quickly.
- Additionally, the bot makes sure that its bid is always slightly lower than the predicted winning bid, adding a safety margin to avoid over-bidding.

.

Variation 3:



```python
from Strategy import StrategyBase
import numpy as np
import xgboost as xgb

class UserStrategy(StrategyBase):

    def __init__(self):
        # Initialize the XGBoost model
        self.model = xgb.XGBRegressor(n_estimators=200, max_depth=6, learning_rate=0.05, random_state=42)
        self.is_trained = False
        self.previous_winners = []
        self.previous_second_highest_bids = []
        self.safety_margin = 0.02  # Ensure profit margin
        self.aggressiveness = 0.1  # Adjust aggressiveness for bidding

    def update_data(self, winner, second_highest_bid):
        """Update historical data with new round results."""
        self.previous_winners.append(winner)
        self.previous_second_highest_bids.append(second_highest_bid)

        # Keep only the last 100 rounds of data
        if len(self.previous_winners) > 100:
            self.previous_winners.pop(0)
            self.previous_second_highest_bids.pop(0)

    def train_model(self):
        """Train the XGBoost model if there is enough data."""
        if len(self.previous_winners) >= 20:  # Require at least 20 rounds of data
            X_train = np.array(self.previous_winners).reshape(-1, 1)
            y_train = np.array(self.previous_second_highest_bids)

            # Train the model
            self.model.fit(X_train, y_train)
            self.is_trained = True
```



```python
def make_bid(self, current_value, previous_winners, previous_second_highest_bids, capital, num_bidders):
    '''This function makes a bid for the auction:
    1. Predicts the second-highest bid using XGBoost.
    2. Ensures that the bid allows for profit while keeping capital in mind.
    3. The strategy minimizes risk for both winner and second-highest bidder.'''
    # Update data with latest round results
    if previous_winners and previous_second_highest_bids:
        self.update_data(previous_winners[-1], previous_second_highest_bids[-1])

    self.train_model()
    # Step 1: Predict second-highest bid using XGBoost
    if self.is_trained:
        predicted_second_highest = self.model.predict(np.array([[current_value]]))[0]
    else:
        predicted_second_highest = current_value * 0.75  # Fallback if not enough data to train the model

    # Step 2: Adjust bid based on predicted second-highest bid and ensure profit
    if capital > current_value * 0.5:
        # More aggressive if capital is high: bid above second-highest prediction
        target_bid = predicted_second_highest * (1 + self.aggressiveness)
    else:
        # Conservative bidding if capital is low
        target_bid = predicted_second_highest * (1 - self.safety_margin)

    # Step 3: Ensure positive profit for the winner
    bid = min(target_bid, capital)  # Bid must not exceed available capital
    bid = max(0, bid)  # Ensure bid is not negative
    bid = min(bid, current_value)  # Ensure bid is not more than player's value

    # Step 4: Adjust to avoid negative profit for second-highest bidder
    if current_value - bid <= 0:
        bid = current_value * (1 - self.safety_margin)

    return bid
```

Overview

Thi strategy leverages XGBoost to predict the second-highest bid, aiming to maximize profitability and minimize risk.

Key Methods

- `update_data(winner, second_highest_bid)`: **Updates historical data with the latest results, keeping only the most recent 100 rounds.**
- `train_model()`: **Trains the XGBoost model using at least 20 rounds of data with** `previous_winners` **as features and** `previous_second_highest_bids` **as targets.**
- `make_bid(current_value, previous_winners, previous_second_highest_bids, capital, num_bidders)`:
  - **Data Update: Incorporates the latest auction results.**
  - **Model Training: Trains the model if there's sufficient data.**
  - **Bid Prediction: Uses the model to estimate the second-highest bid.**
  - **Bid Adjustment:**
    - **Aggressive: If capital is high, bids more aggressively based on predictions.**
    - **Conservative: If capital is low, bids more conservatively.**

    - **Profit Assurance: Ensures bids are positive and avoids negative profit for the second-highest bidder.**

Strategy Suitability for this variation

- **Predictive Modeling: Uses XGBoost to forecast the second-highest bid, improving bidding strategy.**
- **Dynamic Bidding: Adapts bids based on capital and predictions, optimizing risk and reward.**
- **Profit Assurance: Ensures bids are within capital constraints and avoids negative outcomes.**

This class is designed to make adaptive and informed bids, balancing profitability and risk in the auction setting.