Pedro Castro, Coleton Grossman, Ian Keilman

Stat 321

Professor Mathews

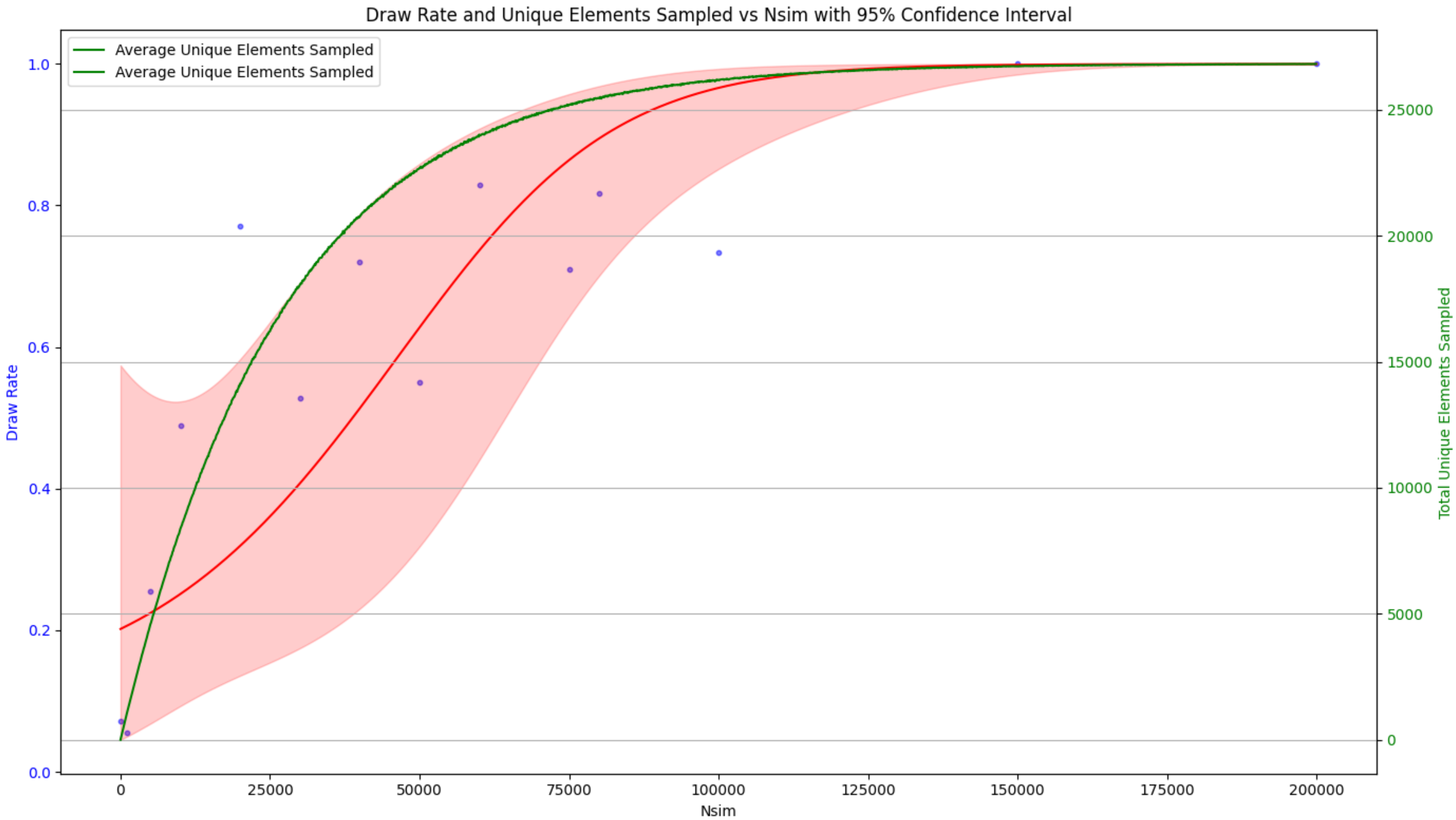
Tic Tac Toe Simulation Study

Our project revolves around the classic game tic tac toe. Tic tac toe is a simple game with a solved play pattern where if each player plays correctly the game will always end in a draw. We want to train a bot to play perfect games. We want to explore characteristics of random forest models as well as running the models on randomly generated data. Is it possible to consistently train a bot given an amount of randomly generated data? If so, at what amount of data can we be confident that we will be able to generate a perfect bot?

We created code to simulate randomly played games of tic tac toe. This data stores game state, so starting with the first move and creating new rows until the game ends with someone winning. This data is guaranteed to not have any resemblance of a strategy, but hopefully at a high enough simulation level, we will start to see bots playing perfectly. Our method of testing was to have our random forest bot play against a perfect bot, which we just found online at https://github.com/anuragjain-git/unbeatable-tictactoe-bot/blob/main/tictactoe.py. If the game ended in a draw, then we successfully made a perfect bot. It is possible to create a perfect rf bot after one game, however this is only possible if the exact perfect game sequence is played, which is 1/26830. 26830 is the number of unique games that can be played, accounting for reflections and rotations. Complications with our simulation setup include repeating data as well as poor play that still results in a win. These issues should hopefully go away after we simulate many games.

Our study starts with running several random games to create a dataset to train a random forest bot. There are *nsim* number of games in each dataset. So, if we have ten different bots at *nsim* level of one hundred thousand games, we simulate one million games. If 5 out of 10 bots draw against our test bot, then one hundred thousand would have a draw rate of 0.5. As we went on, we needed to run more simulations, which required us to store past simulations and add new ones. Although our idea is simple, putting it into code and storing our data was one of the hardest parts of this project.

Before running our simulations, we wanted to make some predictions. We expect our draw rate to start at 0, and end at one. We expect this progress to look something like $f(x) = -1/x$. This intuition comes from what we have done with sampling with replacement. If we can get so many games that we come close to having every game combination, our rf bot should be able to generate a perfect bot every time. This means that at some point diminishing returns will kick in and we will have an asymptote at (hopefully) $y=1$.



AIC for Degree 3: -37.51

We found that our predictions about shape of draw rate by *nsim* level were correct. We start quickly gaining a draw percentage but flatten out before we reach a 100% draw rate. As you can see in this graph, with the red line being a regression line and green being the sampling with replacement line, we have a very wide range for the lower levels of simulation. Including a 95% confidence interval shows that at around 100,000 simulation is when we get a significant difference from 0. Due to long computing times, we ran our larger *nsims* less frequently. If we ran this more, we could get a tighter confidence interval. I also think that we would get a regression line more closely resembling the sampling without replacement, although they both reach 1 at around the same time, which is interesting. It is cool to see that we could consistently get to a draw rate of one, even though it takes about 150000 games to train a bot.

Further testing could include running different types of bots to see what would be most efficient with our data. We could run more simulations to get a more accurate confidence interval. We also speculate that we are not able to truly get a draw rate of one and can only get close to it. This would require more testing at higher *nsim* levels but would possibly be explained by random forest bots not being consistent and having chances of drawing the same games multiple times. We also wonder if we could create a data set containing 26830 combinations of unique games that would further increase efficiency. This seemed like it could easily go wrong, and would have possibly large implications, so we decided not to test this, but it would be interesting to see if we could just replicate our findings.

Here is the GitHub repository with our CVSs and files:

https://github.com/IanKeilman/TTTbot-

Below is our full python code, ran in a Jupiter notebook.

```
# %% [markdown]

# Below is the skeleton code to make the tic tac toe game.


# %%

import random

import math

import numpy as np

import pandas as pd

from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score

import matplotlib.pyplot as plt

from xgboost import XGBClassifier

import os


# %% [markdown]

# making the rules for the winner and draw

#

# https://github.com/anuragjain-git/unbeatable-tictactoe-bot/blob/main/tictactoe.py


# %%
```

```python
# Utility functions
def check_winner(board, player):
    """Check if the player has won the game."""
    winning_combinations = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # Columns
        [0, 4, 8], [2, 4, 6]              # Diagonals
    ]
    return any(all(board[pos] == player for pos in combo) for combo in winning_combinations)


def is_draw(board):
    """Check if the game is a draw."""
    return ' ' not in board and not (check_winner(board, 'X') or check_winner(board, 'O'))


def available_moves(board):
    """Return a list of available moves."""
    return [i for i, spot in enumerate(board) if spot == ' ']


def print_board(board):
    """Print the current board state."""
    print(f"{board[0]} | {board[1]} | {board[2]}")
    print("--+---+--")
    print(f"{board[3]} | {board[4]} | {board[5]}")
    print("--+---+--")
    print(f"{board[6]} | {board[7]} | {board[8]}")


# %%
# State statistics for tracking game outcomes
```

```
state_statistics = {}


def update_state_statistics(state, winner):
    """Update the statistics for a given game state."""
    if state not in state_statistics:
        state_statistics[state] = [0, 0, 0, 0]  # [games_played, x_wins, o_wins, draws]
    state_statistics[state][0] += 1  # Increment games played
    if winner == 'X':
        state_statistics[state][1] += 1  # Increment X wins
    elif winner == 'O':
        state_statistics[state][2] += 1  # Increment O wins
    elif winner == 'draw':
        state_statistics[state][3] += 1  # Increment draws


def simulate_game(agent, verbose=False):
    """Simulate a single game and update state statistics."""
    board = [' '] * 9
    current_player = 'X'
    state_action_history = []

    while True:
        state = tuple(board)
        action = agent.choose_action(board, current_player)
        board[action] = current_player
        state_action_history.append((state, action, current_player))

        if check_winner(board, current_player):
            winner = current_player
```

```python
            for s, a, p in state_action_history:
                update_state_statistics(s, winner=winner)
            break
        elif is_draw(board):
            for s, a, p in state_action_history:
                update_state_statistics(s, winner='draw')
            break
        else:
            current_player = 'O' if current_player == 'X' else 'X'

    if verbose:
        print_board(board)

def train_agent(agent, episodes):
    """Train the agent by simulating games and updating state statistics."""
    for episode in range(episodes):
        simulate_game(agent)
    print(f"Training complete after {episodes} episodes.")

def calculate_win_draw_rates():
    """Calculate the win and draw rates for all encountered game states."""
    win_draw_rates = {}
    for state, stats in state_statistics.items():
        games_played, x_wins, o_wins, draws = stats
        if games_played > 0:
            x_win_rate = x_wins / games_played
            o_win_rate = o_wins / games_played
            draw_rate = draws / games_played
```

```
            win_draw_rates[state] = {
                'X_win_rate': x_win_rate,
                'O_win_rate': o_win_rate,
                'draw_rate': draw_rate
            }
    return win_draw_rates


# %% [markdown]
# Making our dataset of randomly played games


# %%
# Random Agent
class RandomAgent:
    def choose_action(self, board, player):
        """Randomly choose an available action."""
        return random.choice(available_moves(board))


# Generate training data from random games
def generate_training_data(num_games):
    def simulate_random_game():
        board = [" "] * 9
        current_player = 'X'
        game_data = []

        while True:
            move = random.choice(available_moves(board))
            board[move] = current_player
            game_data.append((board[:], current_player))
```

```
            if check_winner(board, current_player):
                return game_data, current_player
            if is_draw(board):
                return game_data, 'draw'

            current_player = 'O' if current_player == 'X' else 'X'

    data, labels = [], []
    for _ in range(num_games):
        game_data, result = simulate_random_game()
        for state, player in game_data:
            features = [1 if cell == 'X' else -1 if cell == 'O' else 0 for cell in state]
            label = 1 if result == player else -1 if result != 'draw' else 0
            data.append(features)
            labels.append(label)

    return np.array(data), np.array(labels)

# Train Random Forest bot
def train_random_forest_bot(num_games):
    data, labels = generate_training_data(num_games)
    X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, random_state=42)
    rf_model = RandomForestClassifier(n_estimators=100, random_state=42, n_jobs=-1)
    rf_model.fit(X_train, y_train)
    y_pred = rf_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Random Forest model accuracy: {accuracy:.2f}")
```

```
    return rf_model

# Random Forest Bot
class RandomForestBot:
    def __init__(self, model):
        self.model = model

    def choose_action(self, board, current_player):
        features = []
        moves = available_moves(board)
        for move in moves:
            temp_board = board[:]
            temp_board[move] = current_player
            features.append([1 if cell == 'X' else -1 if cell == 'O' else 0 for cell in temp_board])
        predictions = self.model.predict(features)
        best_move = moves[np.argmax(predictions)]
        return best_move

# Perfect Bot Logic (Minimax)
def evaluate(board):
    if check_winner(board, 'O'):
        return 1
    elif check_winner(board, 'X'):
        return -1
    else:
        return 0

def minimax(board, depth, alpha, beta, is_maximizing):
```

```
    score = evaluate(board)
    if score == 1 or score == -1 or is_draw(board):
        return score

    if is_maximizing:
        best_score = -math.inf
        for i in available_moves(board):
            board[i] = 'O'
            best_score = max(best_score, minimax(board, depth + 1, alpha, beta, False))
            board[i] = " "
            alpha = max(alpha, best_score)
            if alpha >= beta:
                break
        return best_score
    else:
        best_score = math.inf
        for i in available_moves(board):
            board[i] = 'X'
            best_score = min(best_score, minimax(board, depth + 1, alpha, beta, True))
            board[i] = " "
            beta = min(beta, best_score)
            if beta <= alpha:
                break
        return best_score

def find_best_move(board):
    best_score = -math.inf
    best_move = -1
```

```
    alpha = -math.inf
    beta = math.inf
    for i in available_moves(board):
        board[i] = 'O'
        move_score = minimax(board, 0, alpha, beta, False)
        board[i] = " "
        if move_score > best_score:
            best_score = move_score
            best_move = i
    return best_move

# Play Random Forest Bot vs Perfect Bot
def play_game_rf_vs_perfect(rf_bot):
    board = [" "] * 9
    current_player = 'X'  # Random Forest bot starts

    while True:
        if current_player == 'X':
            move = rf_bot.choose_action(board, current_player)
        else:
            move = find_best_move(board)  # Perfect bot logic

        board[move] = current_player

        if check_winner(board, current_player):
            return current_player
        if is_draw(board):
            return 'draw'
```

```
        current_player = 'O' if current_player == 'X' else 'X'


def simulate_rf_vs_perfect(rf_bot, num_games):
    results = {'rf_wins': 0, 'perfect_wins': 0, 'draws': 0}


    for _ in range(num_games):
        result = play_game_rf_vs_perfect(rf_bot)
        if result == 'X':
            results['rf_wins'] += 1
        elif result == 'O':
            results['perfect_wins'] += 1
        elif result == 'draw':
            results['draws'] += 1


    return results


# Function to create multiple bots, test them, and summarize results
def create_and_test_bots(nsim_levels, bots_per_level, comp_games, existing_bot_ids):
    results = []
    for nsim in nsim_levels:
        # Get existing bot ids for this nsim level
        existing_bots_for_nsim = [bot_id for bot_id in existing_bot_ids if
bot_id.startswith(f"Bot_{nsim}_")]
        # Extract the numbers after 'Bot_{nsim}_' to find existing bot numbers
        existing_bot_numbers = [int(bot_id.split('_')[-1]) for bot_id in existing_bots_for_nsim]
        max_existing_bot_number = max(existing_bot_numbers) if existing_bot_numbers else 0
```

```
    # Start bot numbering from next available id
    for i in range(1, bots_per_level + 1):
        bot_number = max_existing_bot_number + i
        bot_identifier = f"Bot_{nsim}_{bot_number}"

        # Just in case, check if bot_id already exists
        if bot_identifier in existing_bot_ids:
            print(f"Skipping {bot_identifier} as it already exists.")
            continue

        # Train a Random Forest bot on a unique dataset
        rf_model = train_random_forest_bot(nsim)
        rf_bot = RandomForestBot(rf_model)

        # Simulate games against the Perfect bot
        test_results = simulate_rf_vs_perfect(rf_bot, comp_games)

        # Summarize results for this bot
        results.append({
            'nsim': nsim,
            'bot_id': bot_identifier,
            'rf_wins': test_results['rf_wins'],
            'perfect_wins': test_results['perfect_wins'],
            'draws': test_results['draws'],
            'total_games': comp_games
        })

# Create a DataFrame from results
```

```python
    df_summary = pd.DataFrame(results)
    return df_summary


# %%
# Main Execution
if __name__ == '__main__':
    # Set parameters
    nsim_levels = []
    # nsim_levels can be extended as needed


    bots_per_level = 15
    comp_games = 1


    # Check if 'summary_df.csv' exists
    if os.path.exists('summary_df.csv'):
        stored_data = pd.read_csv('summary_df.csv')
    else:
        stored_data = pd.DataFrame()


    # Get existing bot_ids to avoid duplicates
    existing_bot_ids = set(stored_data['bot_id'].unique()) if not stored_data.empty else set()


    # Create and test bots, adjusting bot ids to avoid duplicates
    df_summary = create_and_test_bots(nsim_levels, bots_per_level, comp_games,
existing_bot_ids)


    # Concatenate stored data and new summary
    df_sim_data = pd.concat([stored_data, df_summary], ignore_index=True)
```

```
# Drop duplicate bot_ids
df_sim_data = df_sim_data.drop_duplicates(subset='bot_id', keep='first')

# Save the updated data to 'summary_df.csv'
df_sim_data.to_csv('summary_df.csv', index=False)

# Display concatenated DataFrame
print(df_sim_data)

# Now, calculate the draw rate by nsim level
nsim_values = df_sim_data['nsim'].unique()

# Calculate total bots made at each nsim level
total_bots = df_sim_data.groupby('nsim').size()

# Calculate total draws, RF wins, and Perfect wins at each nsim level
total_draws = df_sim_data.groupby('nsim')['draws'].sum()
total_rf_wins = df_sim_data.groupby('nsim')['rf_wins'].sum()
total_perfect_wins = df_sim_data.groupby('nsim')['perfect_wins'].sum()

# Calculate total games at each nsim level
total_games = df_sim_data.groupby('nsim')['total_games'].sum()

# Calculate rates
draw_rate = total_draws / total_games
rf_win_rate = total_rf_wins / total_games
perfect_win_rate = total_perfect_wins / total_games
```

```
    # Create a DataFrame to consolidate the results
    summary_df = pd.DataFrame({
        'nsim': nsim_values,
        'Total Bots': total_bots.values,
        'Total Draws': total_draws.values,
        'Total RF Wins': total_rf_wins.values,
        'Total Perfect Wins': total_perfect_wins.values,
        'Total Games': total_games.values,
        'Draw Rate': draw_rate.values,
        'RF Win Rate': rf_win_rate.values,
        'Perfect Win Rate': perfect_win_rate.values
    })

    # Ensure all rates are between 0 and 1
    summary_df[['Draw Rate', 'RF Win Rate', 'Perfect Win Rate']] = summary_df[['Draw Rate',
'RF Win Rate', 'Perfect Win Rate']].clip(0, 1)

    # Save the summary DataFrame with draw rates to CSV
    summary_df.to_csv('draw_rates_by_nsim.csv', index=False)

# %%
# Load and filter the data
df_summary = pd.read_csv('draw_rates_by_nsim.csv')
df_summary = df_summary[df_summary['nsim'] <= 200000]

# Define predictor and response variables
X = df_summary[['nsim']].values
```

```
y = df_summary['Draw Rate'].values

# Handle y values exactly at 0 or 1 by applying a small epsilon
epsilon = 1e-4
y = np.clip(y, epsilon, 1 - epsilon)

# Logit transformation
logit_y = np.log(y / (1 - y))

# Fit polynomial regression (e.g., cubic) with covariance matrix
degree = 3
poly_coeffs, cov = np.polyfit(X.flatten(), logit_y, degree, cov=True)
poly_model = np.poly1d(poly_coeffs)

# Predict on training data
y_pred_logit = poly_model(X.flatten())
y_pred = 1 / (1 + np.exp(-y_pred_logit))  # Inverse logit

# Calculate AIC
residual_sum_of_squares = np.sum((y - y_pred) ** 2)
n = len(y)
k = degree + 1  # Number of parameters
aic = n * np.log(residual_sum_of_squares / n) + 2 * k
print(f"AIC for Degree {degree}: {aic:.2f}")

# Generate x values for plotting the polynomial fit
x_fit = np.linspace(X.min(), X.max(), 500)
y_fit_logit = poly_model(x_fit)
```

```
y_fit = 1 / (1 + np.exp(-y_fit_logit))

# ----------------------------
# 2. Calculate 95% Confidence Intervals
# ----------------------------

# Create the design matrix for polynomial terms
X_fit = np.vander(x_fit, degree + 1)

# Calculate the standard error of the predictions
# The variance of the prediction is X_fit @ cov @ X_fit.T for each x_fit
# Since X_fit is (500, degree+1) and cov is (degree+1, degree+1),
# the variance for each prediction is the diagonal of X_fit @ cov @ X_fit.T
pred_variance = np.sum(X_fit * (X_fit @ cov), axis=1)
pred_std_error = np.sqrt(pred_variance)

# 95% confidence interval using the standard normal distribution
confidence_level = 1.96  # for 95% confidence
y_fit_upper_logit = poly_model(x_fit) + confidence_level * pred_std_error
y_fit_lower_logit = poly_model(x_fit) - confidence_level * pred_std_error

# Apply inverse logit to get confidence intervals on the original scale
y_fit_upper = 1 / (1 + np.exp(-y_fit_upper_logit))
y_fit_lower = 1 / (1 + np.exp(-y_fit_lower_logit))

# ----------------------------
# 3. Perform Unique Elements Sampling Simulation
# ----------------------------
```

```
N = 26830  # Total unique elements
max_samples = 200000  # Maximum samples to draw
step = 100  # Step size for plotting
iterations = 4  # Number of simulations for averaging

unique_counts = []
sample_sizes = range(1, max_samples + 1, step)

for n in sample_sizes:
    total_unique = []
    for _ in range(iterations):
        sampled = np.random.choice(N, n, replace=True)
        total_unique.append(len(set(sampled)))
    unique_counts.append(np.mean(total_unique))

# Save the unique counts to a CSV file
unique_counts_df = pd.DataFrame({
    'nsim': list(sample_sizes),
    'Total Unique Elements Sampled': unique_counts
})
unique_counts_df.to_csv('unique_counts.csv', index=False)

# ----------------------------
# 4. Combine Both Plots on the Same Chart with Confidence Intervals
# ----------------------------

plt.figure(figsize=(14, 8))
```

```
# Plot Draw Rate and its polynomial fit on the primary y-axis
plt.scatter(X, y, color='blue', label='Original Draw Rate Data', alpha=0.5, s=10)
plt.plot(x_fit, y_fit, color='red', label=f'Polynomial Fit (Degree {degree})')
plt.fill_between(x_fit, y_fit_lower, y_fit_upper, color='red', alpha=0.2, label='95% Confidence Interval')

plt.xlabel('Nsim')
plt.ylabel('Draw Rate', color='blue')
plt.title('Draw Rate and Unique Elements Sampled vs Nsim with 95% Confidence Interval')
plt.tick_params(axis='y', labelcolor='blue')

# Create a secondary y-axis for the Unique Elements Sampled
ax2 = plt.gca().twinx()
ax2.plot(sample_sizes, unique_counts, color='green', label='Average Unique Elements Sampled')
ax2.set_ylabel('Total Unique Elements Sampled', color='green')
ax2.tick_params(axis='y', labelcolor='green')

# Combine legends from both y-axes
lines_1, labels_1 = plt.gca().get_legend_handles_labels()
lines_2, labels_2 = ax2.get_legend_handles_labels()
plt.legend(lines_1 + lines_2, labels_1 + labels_2, loc='upper left')

plt.grid(True)
plt.tight_layout()
plt.show()
```