RISK Data Project Fall 2020

Jacob Murri, Joseph Wilkes, Caleb Wilson, & Lukas Erekson

December 10, 2020

1 Research Question and Data Overview

The goal of this project is to predict which player will win a game of RISK given only the state of the board at a given turn. In order to train a model to make such predictions, we forked and modified a RISK game engine written in Python called PyRisk [1]. PyRisk contains several different automated RISK playing computer agents (referred to hereafter as AIs) which use different strategies to play RISK. We used PyRisk to run tens of thousands of games pitting these AIs against each other. With this dataset, we developed several models that predict the winner based on the state of the board at a given turn.

All code used to generate, parse, and feature-engineer the data used in this project is available on GitHub at https://github.com/LukasErekson/pyrisk. Due to the large quantity of game data, this data is not available on GitHub.

The rest of this section describes some specific features of the PyRisk engine, previous work, and our data collection process. In Section 2, we discuss how we cleaned the data and the features we engineered. Sections 3 and 4 describe the various ways we visualized the data and applied machine learning algorithms to analyze the data, respectively. Finally, we evaluate the ethical implications of our project in Section 5, and present our conclusions in Section 6.

1.1 Description of PyRisk

The PyRisk engine implements a simplified version of the board game RISK and allows for games where initial territories are either dealt randomly or claimed by players. For readers who need an overview of basic RISK gameplay, see [2], pages 1-9. We also note that PyRisk makes a few simplifications to standard RISK gameplay; for instance, it does not support troop bonuses from cards. For a more exhaustive overview of the rules of RISK, see [3]. Although much of the functionality RISK was already included, we needed to make a few modifications to the data before performing analysis. For a detailed description of the modifications that were made then see the `README` on the Github repository.

1.2 Previous Work

Because of the popularity of RISK, in the past others have sought to use learning methods to answer questions similar to those we are asking. For example, Jacob Munson attempted to create a neural net which predicts the winner of two-player RISK games; his work is found in [2]. His classifier performed relatively well, and he used his results to give recommendations for good RISK gameplay. Although Munson uses different methods and considers different types of games (he focuses on two-player games rather than three-player or six-player games as we do), we were inspired by the way that he stored the state of the RISK game board in a tabular format. This enabled us to effectively

store and analyze our data. We were also able to gain inspiration from how well his classifier worked and how it was able to identify important strategic elements of the game.

1.3 Data Collection

Since we could not find any open datasets which contained RISK game data, we had to use the PyRisk engine to create our own dataset. The major technical downside of using PyRisk was that the engine did not export any tabular data from an individual game. It did, however, export a log file in `.txt` format that recorded the state of the game on an attack by attack basis. We wrote a parser to get the relevant data from the log file in tabular form. Because of the script, the data collection and cleaning process is fairly robust; as long as the game data has the same basic form as what PyRisk outputs, it should easily create DataFrames for any game of RISK. A more thorough description of the data cleaning process is included in section 2.

We ran thousands of games of RISK where each player is an AI that has various levels of skill and strategy implemented into their play style (see B). We modified PyRisk to run multiple games in parallel to speed up our data collection. An example of this log file is displayed in the appendix in Figure 9.

Our parser uses regular expressions to convert each game's log file into a Pandas DataFrame where each row corresponds to the state of the board at a given turn and each column corresponds to a feature of the game during that turn. For the structuring of our data, we take after the ideas of Munson's *Unit List*, [2] pages 30-31: the state of the board is contained in $42 \times n$ columns (n being the number of players) which list the troop count of each player in each territory (see Figure 1). The DataFrame also has a column indicating which player won the game, got second place, third place, etc. Using this structure, we engineered several features which we hypothesized to have predictive power. More detail about these features can be found in Section 2.2, and a sample from the rows and columns of one of the game DataFrames we used can be found in Figure 1. Other summary data which did not lend itself well to inclusion in these DataFrames, such as the types of AI in each game, was stored in a dictionary.

Player 3 Alaska	Player 3 Northwest Territories	Player 3 Greenland	Player 3 Alberta	Player 3 Ontario	winner
1.0	1.0	67.0	1.0	1.0	3
1.0	1.0	67.0	1.0	1.0	3
1.0	1.0	67.0	1.0	1.0	3
1.0	1.0	67.0	1.0	1.0	3
85.0	1.0	1.0	1.0	1.0	3

Figure 1: A tail from a DataFrame corresponding to one game (prior to our feature engineering). Only 6 out of the 253 columns are shown. Here, we can see during Player 3's last turn, they moved 84 troops into Alaska and moved 66 troops away from Greenland.

Using the parser that we wrote and the PyRisk AI, we generated tens of thousands of games and quickly realized that we had too much data to train our models on with the computational power we had access to. Because we wanted to include the greatest possible diversity of games, we decided to thin out the dataset by only including data from specific turns (we called these kth-turn

datasets). By using combined kth-turn data, rather than the full dataset of all turns in every game, it became computationally feasible to build models that would predict the winner of the game given the state of the board at any turn, as well as models which were optimized for one specific turn. For the former types of models, we were able to analyze accuracy at various turns (see Figure 7).

1.4 On the Data's Validity

We recognize that our model may accurately predict winners of games where the players are all one of the PyRisk AI types, but it may perform poorly when trying to predict the outcome of a game with different types of AI or human players. This weakness stems from all our data being generated by combinations of the specific PyRisk AI. Although it may not have bearing on games with different types of players, we believe that our analysis still has the potential to identify key features that correlate with victory or defeat in RISK.

2 Data Cleaning and Feature Engineering

2.1 Data Cleaning

Although we generated our own data – giving us almost total control of its structure– we did run into some issues of missing data. It's theoretically possible for a game of RISK to last indefinitely, so to keep game lengths reasonable, we ended games that went longer than $120 * n$ turns (n being the number of players) without a winner and recorded them as stalemates. Additionally, games that were interrupted during play would output an incomplete log file, failing to identify a winner or the game as a stalemate. Since we're solely interested in calculating the probabilities that a given player will win, we dropped stalemates and incomplete games from our dataset completely.

When we initially used the engine, AlAI was not functioning and would crash the game so we could not generate games with it. We fixed AlAI so we could include it as a player in the game data we generated. Descriptions of the AI's can be found in B.

2.2 Feature Engineering

As explained above, the data we took from the log files only included information about which players held which territories. In order to help our classifier predict, we created several new features which roughly correspond to common heuristic strategies for playing RISK. The following is a list of new features that we developed, many of which utilized the underlying graph structure of RISK. (In the list, a *boundary territory* refers to a territory which borders the territory of another player.)

- Country count: Each player's total number of territories held
- Troop count: Each player's total number of troops
- Continental control: Whether or not each player holds a specific continent
- Troop reinforcements: The total number of troops a player will receive in their next turn
- Cut edges: The number of boundary edges which cross into each player's territory
- Boundary nodes: The number of each player's boundary territories
- Boundary fortifications: The total number of troops on each player's boundary territories
- Boundary territory percentage: Each player's ratio of boundary territories to total territories
- Boundary troop percentage: The proportion of each player's troops on boundary territories

- Boundary fortifications: Each player's average number of troops in boundary territories
- Connected components: The number of connected components in each player's possession
- Final placement: The winner, second place player, etc.

3 Data Visualization and Basic Analysis

3.1 A Single Game

To give the reader an initial feel for the data, we plot a couple important features over the course of a single six-player game which features three different PyRisk AIs (see B for the descriptions of each AI). This allows us to identify trends within a single game that may provide insight into how each player will likely place. Afterwards, we examine some interesting visualizations of several thousand games.

For these visualizations, we do not include the 150 turns taken for the initial set up (1 turn per troop placement for 25 troops for six-players). All of the data generated (including the game we are looking at here) have randomly dealt initial territories anyway, making it not very insightful to examine these initial 150 turns. We start indexing at 0, which is the state of the board right after setup, with 1 being the first turn a player takes etc.

The features that we're visualizing in Figure 2 are the total number of countries or territories each player has and the total number of troops. These features were selected because they seemed like natural measures that correlate with which player ends up winning the game. As the visualizations demonstrate, the features plotted over time give clear indications of each player's final placement in the game.

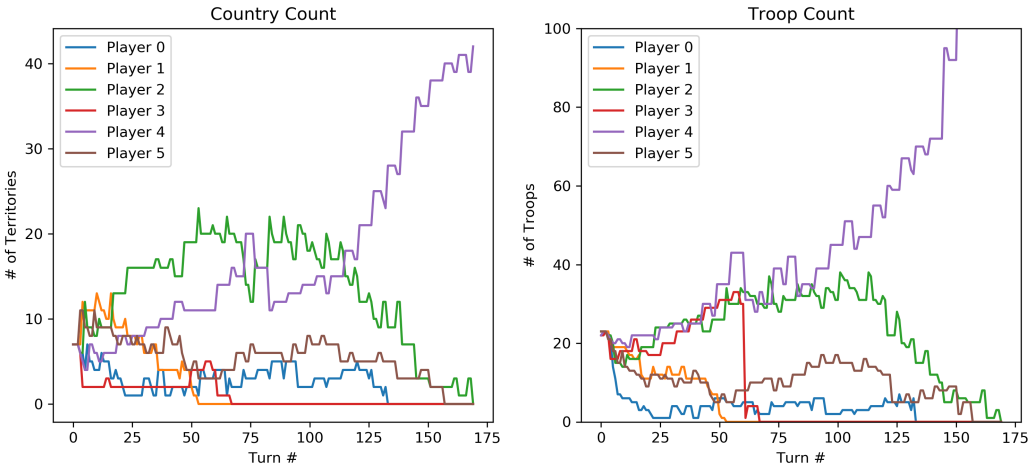

Figure 2: Depiction of country count (left) and and troop count (right) for each player throughout a single game. Around turn 125, it becomes obvious that Player 4 will win because their troop count and country count quickly surpasses and overwhelms that of the other players.

3.2 Basic Analysis

Upon initial analysis of our data we discovered there was a significant unevenness pertaining to the percentage of wins per player. This bias is illustrated in Figure 3. This was due to games with four

ChronAI and two StupidAI being produced so much more quickly and in greater quantity than other games. For the machine learning analysis, we used a subset the data with an even distribution of wins to overcome this bias.

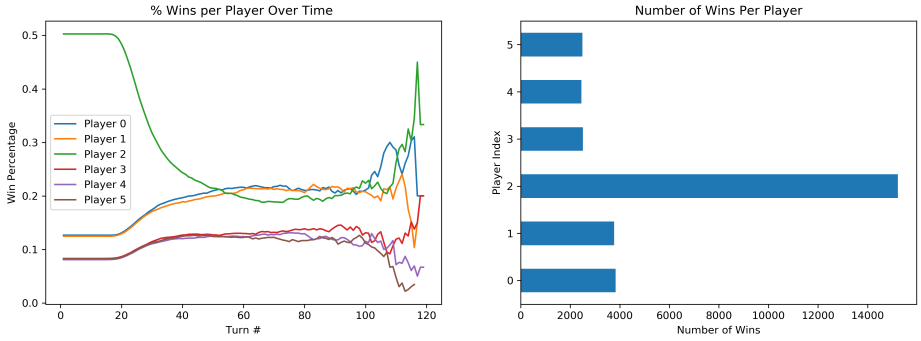

Figure 3: Depiction of number of wins per player.

To understand the data better, we looked at average game duration, as displayed in Figure 4. The mean duration of a six-player game is 207 turns. This insight influenced our decision of how to create one dataset for multiple models that predicted on games over time. We wanted our test set to include games that lasted longer than the mean duration. This is described more thoroughly in the section 4.1.

We applied the t-SNE algorithm as well which can be seen in Figure 5. Notably, t-SNE cannot fully separate the data into clusters but at high perplexities, it does a good job of creating star-like jets for each player. This phenomenon can be explained by the fact that t-SNE ran on a dataset that included many different points in time. Games early on probably look quite similar regardless of which player wins, but when the eventual winner gets closer to winning, the games start to look quite different.

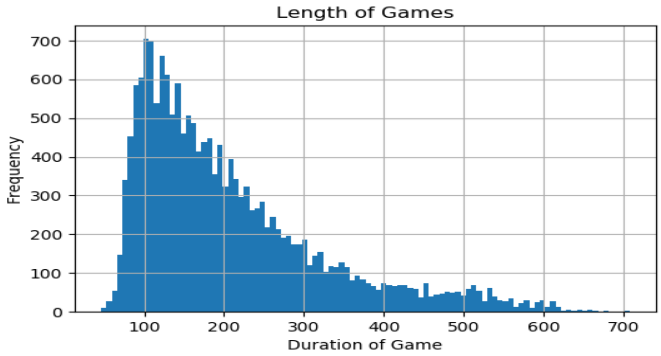

Figure 4: Distribution of game duration in our dataset. The mean duration of a six-player game is 207 turns.

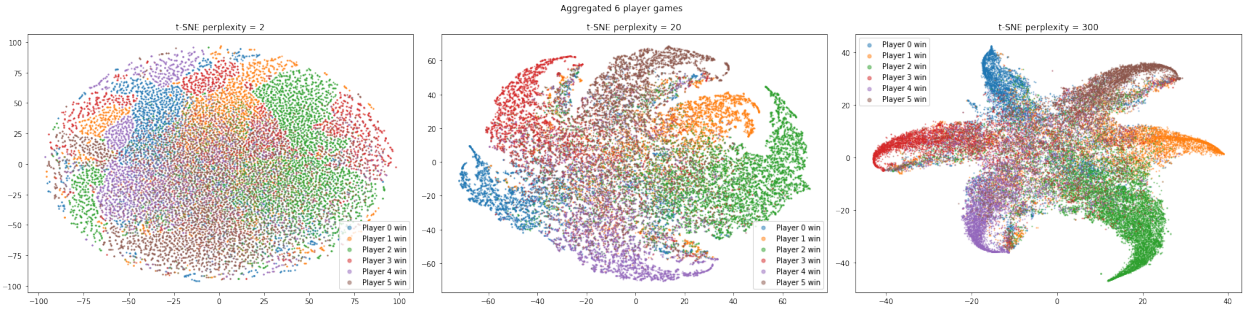

Figure 5: Several t-SNE visualizations of the combined kth-turn data which was used to train our logistic regression, random forest, and AdaBoost models.

4 Learning Algorithms and In-depth Analysis

In this section, we discuss which learning algorithms we chose to use and give details about their parameters and performance. We measured the performance of the models using the methods described in Section 4.1. We conclude with a discussion on feature importance and how three-player games compare to six-player games according to our analysis.

We focused on logistic regression, random forests, boosted decision trees, and the k-nearest neighbor algorithm to create our classifiers. Although linear regression is commonly used in machine learning, we wanted our model output to correspond to the probability that each player would win, so we felt that linear regression with its unbounded (and possibly negative) output parameters did not align well with our goals for this project.

We built two types of models: models which predict the winner of the game given the state of the board at any turn and models optimized for one specific turn. In general, models which were optimized for one specific turn (for example, models which were trained on the 200th turn dataset only) achieved higher accuracy than models trained on the combined kth-turn datasets. However, we believe this is due to overfitting. For comparison purposes, we focused on models which were trained on the combined kth-turn datasets.

We performed logistic regression with elastic net regularization and found fairly strong initial results. As we engineered more features and generated more data, we were able to fine-tune the hyperparameters. An example of the confusion matrix for all of the kth-turn datasets is given below in Figure 6.

Due to time constraints, we could only run AdaBoost on the whole dataset and not XGBoost. However, an XGBoost model trained on the 300th turn dataset achieved about 70 % accuracy.

Our random forest model consisted of 200 decision tress with a max depth of 12 and a maximum of 51 features for each split. Performance of our random forest model can be seen below in Figure 6 and Figure 7.

We also employed a k-nearest neighbors classifier (KNC) to see how well it could predict the games. First, we used Principle Component Analysis (PCA) to reduce the dimensions of the data before running the KNC. A grid search revealed the optimal number of components for PCA to be 60 and the optimal number of nearest neighbors for the KNC to be 30. Our KNC performed comparably to our logistic regression and random forest models, as can be seen in its confusion matrix in Figure 6.

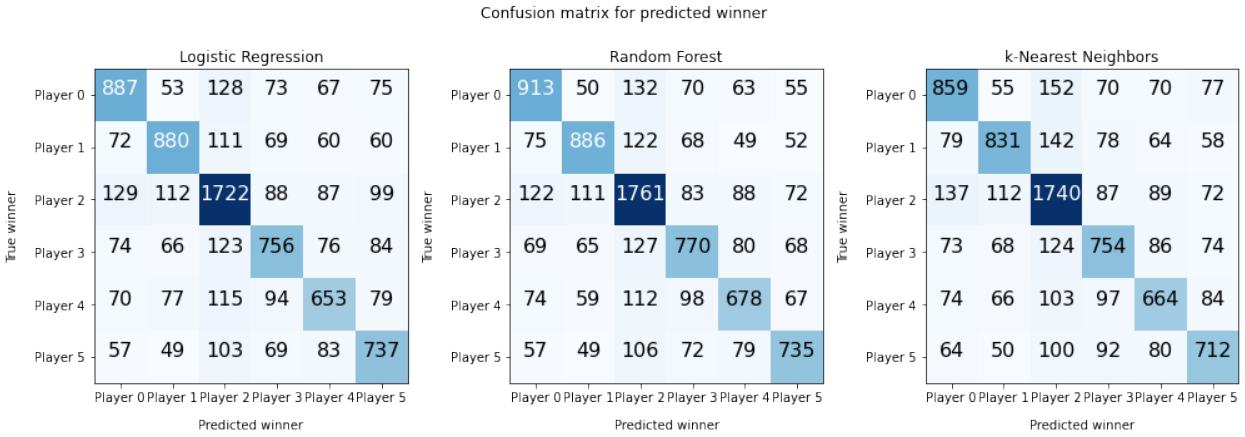

Figure 6: A comparison of the confusion matrices for a multinomial logistic classifier and a random forest classifier trained on the combined kth-turn datasets. This shows the number of games the predictors classified both correctly and incorrectly. A perfect classifier would have a confusion matrix with a dark diagonal stripe and no predicted games outside of the diagonal. The features used in this regression were player continent control, continental reward, country count, total reinforcements, troop count, and troop increase due to country count. The target was which player won. Clearly, logistic regression, the random forest, and the k-nearest neighbors classifier yield very similar results.

4.1 Model Comparison

Given that we tried to use many models as classifiers for our data, we wanted to find out which model was best. In order to do so, there were two questions that we needed to answer:

1. How do different models compare in predictive accuracy?
2. How does that predictive capability change when tested at various stages of the game?

Here we describe how we attempted to answer these questions for six-player games.

To compare different prediction models, the models had to be trained and tested on the same data. These are the steps we followed in order to analyze the data:

1. We created a test set (1,500 games) from the kth turn datasets. We selected games which lasted between 250-300 turns. Although this was an arbitrary choice, we made it so that the test games didn't finish too early nor last too long. It is possible that this test set was biased by unintentionally selecting for games of a certain type or games for which a certain AI combination was playing. Although this potential quandary can't be resolved with current time constraints, it will be worth investigating in the future.
2. We built the training set (50,000 games). To build the training set we used data that wasn't included in the test set. However, as described in section 3.2, the data had an apparent unevenness in the distribution of player winnings. Therefore, we sampled from these remaining games so that the distribution of which players won was uniform.
3. We performed a hyperparameter grid search for each model. After optimizing the hyperparameters, then we were able to compare the best versions of each model with each other.

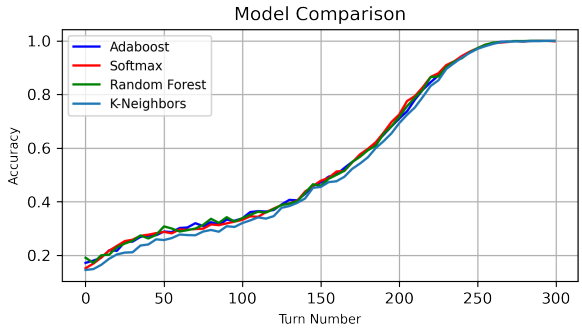

Figure 7: Comparison of several models' predictive capability across time for six-player games. Each of these models was optimized with a grid search.

As seen in the Figure 7, the models achieve a very similar accuracy despite being constructed in completely different ways. One question this raises is why the models produced such similar accuracies. One hypothesis is that our models have reached some sort of common accuracy threshold, and that it may not be possible to predict across all turns with greater accuracy due to the inherent randomness in the game RISK. We are still looking for a model which might be able to break through this apparent threshold.

4.2 Feature Importance

A natural question is what features tend to be the most important in predicting the winner of the game. Using our random forest models, we calculated the OoB feature important scores for our dataset. In order, the most important features were total reinforcements, the percent of territories that were boundary territories, troop counts, and whether or not the players had control of Australia. To understand better just how important these features were, we trained a series of logistic regression models with these features permuted and noted how the loss function was subsequently changed. The loss function was most affected by troop counts, total reinforcements, Australia control, and percent of boundary territories, in that order.

What features are the least important when it comes to predicting the winner? Again looking at the OoB importance scores, the features that always came last were the features corresponding to continent control of South America, Europe, Africa, and Asia, in decreasing order of importance. It's interesting to note that continent control of North America and Australia consistently scored much higher than the other continents. This surprised us because continental control of South America requires the capture of 4 territories and the defence of 2 boundary nodes whereas North America requires the capture of 9 territories and the defence of 3 boundary nodes. Although it's theoretically easier to conquer South America than North America, control of South America correlates less with winning.

Of course, more strategy analysis (which is beyond the scope of this project) needs to be done before making any definitive conclusions about *how* these features correlate with winning, but these results are somewhat surprising.

We also looked into applying principal component analysis (PCA) to our data to see if projecting the data onto its first two or three principal components resulted in any sort of meaningful visualization or clustering. However, low-dimensional PCA projections failed to yield any interesting insights into the data.

4.3 Three-Player Games – A Subset or A Game-changer?

While most of our analysis was focused on games with six-players, we were also interested in the difference between three-player and six-player games. We found that our models trained individual kth turn datasets of three-player games performed as well (in terms of accuracy) as our models trained on six-player games. For example, logistic regression models correctly predicted the winner with approximately 50 % accuracy starting on the 10th turn of play and got more accurate as the turns went on. Due to time constraints, we did not aggregate the three-player datasets like we did the six-player datasets.

Additionally, the features which were most important for six-player games were also the most important for three-player games. Despite the different number of players, it seems that the important indications of victory at a given point in the game are the same: how many troops does a player have (troop count), are they placed on boundary territories, and how much is the troop count increasing (total reinforcements). Unfortunately, time constraints didn't allow for a deeper comparison of these two different types of gameplay.

5 Ethics

We have thought deeply about the ethical implications of this analysis. In order to explain why we believe our work is ethical, we have evaluated it with respect to the five metrics listed by Patil et. al in "Data Science and Ethics" [4]:

1. Consent: All of the data that we use for our project is computer-generated, so concerns related to user data are nonexistent. PyRisk was developed and released publicly as open source software on GitHub, so we are sure that using their code is within their consent.
2. Clarity: We attempt to be very clear about what our model can and cannot accomplish. While our model can attempt to predict the outcome of a game, it is not always accurate, and it is only trained on computer-generated games. Because of the limited scope of our training data, our model may not perform as well on games where the players are human or games which use slightly different versions of RISK rules (like including troop bonuses from cards).
3. Consistency and Trust: Our model does not use any human-generated data. Therefore, anybody who uses our model can be confident that their data is not being used at all, let alone being harvested or used for malicious purposes. If we expanded our model to include human-generated games, we would make sure to only include games for which all players had given us permission to access the game data. We would also make sure to only use the data in the ways indicated by the users.
4. Control and Transparency: Many machine learning models are relatively opaque in the sense that it is difficult to tell what the model is doing under the hood. Others are hidden, in the sense that the author has not published the specifications of the model for others to see. Both of these types of models lead to ethical issues because trusting that the model does what it says it does requires trusting the author as well. But our model does not fall into either of these categories. Our model code is open-source and published on GitHub for anyone to see. Anybody who wants to understand how we parsed and analyzed our data is welcome to do so. This helps ensure that others can independently verify the accuracy of our claims and even improve upon our modeling choices if they see a better path forward.

5. Consequences: Fortunately, being able to win a game of RISK does not lead to actual world-domination, so we can rest assured that our research says nothing about how to take over the world. As noted in the "Clarity" bullet point above, our model is not always accurate, and – as far as we can tell– does not have enough predictive power to be useful in any sort of professional RISK competition. Furthermore, because the model does not suggest a specific course of action for one player, it cannot be used for cheating in a high-stakes situation. However, the insights gained from the model can benefit casual players of RISK by revealing which features and aspects of the game are most important. Thus, it can be used to help develop an understanding of the game but will not have adverse consequences related to cheating.

As a last ethical point of consideration, we can evaluate whether our model can result in destructive feedback loops. Because our model cannot play the game RISK, it is never trained on the output of its own data. Hence there is no feedback loop that results from the use of our model. If we extended our current work to create an AI which plays the game of RISK, we would have to carefully consider the implications of iteratively training the model on the data that it has generated. However, in its current state, we are confident that our model will not generate any feedback loops for human players or have any adverse consequences.

6 Conclusion

The board game RISK is inherently stochastic. The winner of the game and the outcome of any individual battle are highly influenced by the roll of the die. In the case of our datasets, even the initial setup is random because the territories that the players start with are randomly dealt. The inherent randomness in RISK makes predicting the winner a nontrivial task, especially given only the current state of the board.

In many cases, our models successfully predicted the winner of the games in our test set. As expected, our models did not do very well at predicting the winner at the beginning of a game, but their accuracy increases as the game progresses. At a certain point in the game, several models could predict almost perfectly which player would win.

As we worked on this project, we came to several unexpected conclusions. First, there wasn't a substantial difference between performance of different models. This could have been a result of any of the following reasons: having too small of a grid search for the hyperparameters, reaching a predictability threshold, or even biased data (due to the unevenness of which types of AIs played in our dataset). Second, three-player games had the same critical features as six-player games. Both of these results surprised us, and with more time, we would be able to investigate this more thoroughly.

We identified a few features that were critical for predicting the winner. These include troop reinforcement, the percentage of territories that were on the boundary of the players controlled region, and troop count. One interesting result identified by the models was the advantage gained by capturing North America over South America. We believe this is because holding North America guarantees the player three more troop reinforcements every turn than holding South America, while only having to maintain one additional boundary territory.

All of the data we generated had AI players and didn't include any human data. Thus, applying our model to human data could result in a decrease in prediction accuracy. We claim however, that our models have identified some of the primary components associated with winning and that if applied a human game of RISK, our model may still have a good chance of predicting the winner.

References

[1] leoandeol. Pyrisk, 2018.

[2] Jacob Munson. Neural network predictions of a simulation-based statistical and graph theoretic study of the board game risk, 2017.

[3] UltraBoardGames.com. How to play risk, 2020.

[4] Hilary Mason Mike Loukides and DJ Patil. *Ethics and Data Science*. O'Reilly Media, 2018.

Appendix

A Extra Visuals

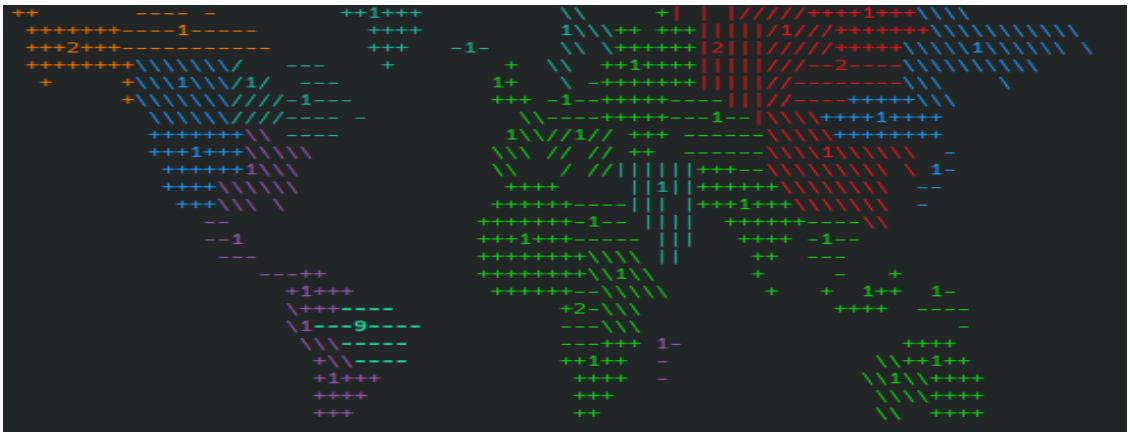

Figure 8: The RISK game board as displayed by PyRisk using Python's `curses` library. Each color corresponds to a player, and territories are distinguished by different symbols (i.e – +, −, /, \, |).

```
INFO:pyrisk:[6, 73, 'Player Areas', 'P;ALPHA;BetterAI', '[]']
INFO:pyrisk:[6, 73, 'Player Areas', 'P;BRAVO;StupidAI', "['North America', 'South America', 'Africa', 'Asia', 'Australia']"]
INFO:pyrisk:[7, 74, 'reinforce', 'P;BRAVO;StupidAI', 'T;Ukraine', '10']
INFO:pyrisk:[7, 74, 'reinforce', 'P;BRAVO;StupidAI', 'T;Northern Europe', '7']
INFO:pyrisk:[7, 74, 'reinforce', 'P;BRAVO;StupidAI', 'T;Southern Europe', '2']
INFO:pyrisk:[7, 74, 'reinforce', 'P;BRAVO;StupidAI', 'T;Great Britain', '3']
INFO:pyrisk:[7, 74, 'reinforce', 'P;BRAVO;StupidAI', 'T;Iceland', '5']
INFO:pyrisk:[7, 74, 'reinforce', 'P;BRAVO;StupidAI', 'T;North Africa', '5']
INFO:pyrisk:[7, 74, 'conquer', 'P;BRAVO;StupidAI', 'P;ALPHA;BetterAI', 'T;Ukraine', 'T;Scandanavia', '(16, 9)', '(1, 9)']
INFO:pyrisk:[7, 74, 'conquer', 'P;BRAVO;StupidAI', 'P;ALPHA;BetterAI', 'T;Northern Europe', 'T;Western Europe', '(10, 7)', '(1, 3)']
INFO:pyrisk:[7, 74, 'elimination', 'P;BRAVO;StupidAI', 'P;ALPHA;BetterAI']
INFO:pyrisk:[7, 74, 'State of the Board', 'P;ALPHA;BetterAI', '{}']
INFO:pyrisk:[7, 74, 'State of the Board', 'P;BRAVO;StupidAI', "{'Alaska': 1, 'Northwest Territories': 1, 'Greenland': 1, 'Alberta': 2, 'Ontario': 1, 'Quebec': 1, 'Western United States': 1, 'Eastern Uni
ted States': 2, 'Mexico': 1, 'Venezuala': 1, 'Brazil': 1, 'Peru': 1, 'Argentina': 2, 'North Africa': 7, 'Egypt': 1, 'East Africa': 1, 'Congo': 1, 'South Africa': 4, 'Madagascar': 2, 'Iceland': 6, 'Great
 Britain': 4, 'Scandanavia': 9, 'Ukraine': 1, 'Northern Europe': 1, 'Western Europe': 3, 'Southern Europe': 7, 'Middle East': 3, 'Afghanistan': 3, 'India': 1, 'South East Asia': 1, 'China': 1, 'Mongolia
': 1, 'Japan': 1, 'Kamchatka': 1, 'Irkutsk': 1, 'Siberia': 2, 'Ural': 2, 'Indonesia': 1, 'New Guinea': 5, 'Eastern Australia': 1, 'Western Australia': 3}"]
INFO:pyrisk:[7, 74, 'Player Areas', 'P;ALPHA;BetterAI', '[]']
INFO:pyrisk:[7, 74, 'Player Areas', 'P;BRAVO;StupidAI', "['North America', 'South America', 'Africa', 'Europe', 'Asia', 'Australia']"]
INFO:pyrisk:[8, 75, 'victory', 'P;BRAVO;StupidAI']
```

Figure 9: This is what the end of a log file looks like. This file had to be parsed into a tabular format.

B Description of PyRisk AI

The PyRisk engine has 4 main different types of AI that follow different strategies:

1. StupidAI: Plays a completely random game, randomly choosing and reinforcing territories, and attacking wherever it can without any strategic considerations.
2. ChronAI: Changes strategy based on whether it considers itself the strongest, intermediate, or weakest player. For example, when the AI believes it is strongest, it will play safely by using strong walls and cautious attacks, and when it is weaker, it will attempt spoiling attacks and target weaker players.
3. BetterAI: An AI that plays better than randomly attacking. It randomly assigns a priority to each continent, and then attempts to gain control and reinforce the continents in that order, slowly expanding its territories.
4. AlAI: An AI similar to BetterAI but with a fixed continent priority order: ['Australia', 'South America', 'North America', 'Africa', 'Europe', 'Asia'].

C Parser Script

This script was the genesis of our data cleaning process. We wrote this process; it allowed us to parse an arbitrary number of the log files which had differing sizes, and contained structures of varying lengths. It helped us turn a text document into a tabular dataset. We were able to do our feature engineering during the parsing as well. It is extremely robust and versatile.

Note that several of the functions called in this script are not included for brevity. For the full script and other code used in this analysis, see https://github.com/LukasErekson/pyrisk.

```
import numpy as np
import pandas as pd
from glob import glob
import h5py
import re
import sys
import os, os.path
from world import T_INDEX
from graph_features import get_graph_features

"""Draft of a log parsing script. This should make it easy to take a log
file and create the data types and files we want for each game.
"""
#turn to false in vim, before running on big file directory
debug=False
log_file_format = '.txt'
log_file_format = '.log'

AREA_INDEX = {'North America': 0,
              'South America': 1,
              'Africa': 2,
              'Europe': 3,
              'Asia': 4,
              'Australia': 5}

def troop_income_due_to_country_possesion(s):
    """
    Get the portion of troop income pertaining to country count

    Parameters
        s (int): the number of countries the player has

    Returns:
        n (int): number of troops to receive next turn
    """
    # if a player has no countries they will receive no incoming troops
    if s == 0:
        return 0
    # each player receives at least 3 troops per turn, must have 12
    # countries or more to get 4+ troops
    if s < 12:
        return 3
    else:
        return s // 3

def get_board_names_for_players(num_players):
    """ Similar to list_player_countries, this returns a 1d list
    of all the countries from n players in order, thus robustly
    selecting the columns that describe the state of the board. This
```

```
    function is useful for carefully choosing the input for graph features

    Output is similar to the array below (which is for a 6 player game):

    array(['Player 0 Alaska', 'Player 0 Northwest Territories',
       'Player 0 Greenland',

       ....

       ,'Player 5 New Guinea',
       'Player 5 Western Australia', 'Player 5 Eastern Australia'])


    Parameters:
        num_players (int): 2 \leq num_players \leq 6

    Returns:
        names_of_player_countries ((42*n,) ndarray): names of columns describing state of the game

    """

    names_of_player_countries = np.array(list_player_countries(player_num=0))
    for i in range(1,num_players):
        names_of_player_countries = np.hstack((names_of_player_countries, list_player_countries(player_num=i)))

    return names_of_player_countries

if __name__ == "__main__":
    # Take in an argument for the file name and ouput file. If none is
    # specified, use the defaults.
    num_args = len(sys.argv)
    if num_args == 1:
        input_dir = 'logs'
    else:
        input_dir = sys.argv[1]
    if num_args == 4:
        output_dir = sys.argv[2]
        output_file = sys.argv[3]
    else:
        output_dir = 'default_log_data'
        output_file = '/parsed_log'
    # Create the output folder if it doesn't already exist
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)
    files_to_parse = glob(input_dir + '/**/*' + log_file_format, recursive=True)
    if debug:
        print('# of files to parse',len(files_to_parse))

    for k, filename in enumerate(files_to_parse):
        if "win_summary" in filename:
            #skip win summaries because they are formatted differently
            continue
        file = ""
        with open(filename, 'r') as fi:
            file = fi.read()
        # Get the list of the players and map them to integers
        p_list_pattern = re.compile("\[0, 0, 'players'.*\n")
```

```
        p_list_str = re.findall(p_list_pattern, file)[0]
        # Player name pattern
        p_name_pattern = re.compile("P;[A-Z;a-z;_]+")
        p_name_list = re.findall(p_name_pattern, p_list_str)
        # Dictionaries mapping players to indicies and vice versa
        player_index = {}
        index_player = {}
        for i, player in enumerate(p_name_list):
            player_index[player] = i
            index_player[i] = player

        num_players = len(p_name_list)
        # Get the winner of the game
        try:
            winner_pattern = re.compile("([0-9]+), 'victory', '(P;[A-Z;a-z;_]+)'")
            total_turns, winner = re.findall(winner_pattern, file)[0]
            total_turns = int(total_turns)
        # No winner, stalemate
        except IndexError:
            try:
                stalemate_pattern = re.compile("([0-9]+), 'Stalemate'")
                total_turns = re.findall(stalemate_pattern, file)[0]
                winner = "None"
                total_turns = int(total_turns)
      # Game never finished (interrupted maybe?) Skip parsing.
            except Exception as e:
                print(filename, e)
                continue
        # Player areas after each turn
        player_area_pattern = re.compile("([0-9]+), 'Player Areas', '([A-Z_a-z;]+)
', ['\"]\[(.*)\]['\"]\]")
        p_areas = re.findall(player_area_pattern, file)
        # Initialize empty area lists
        Area_lists = np.zeros((6, num_players, total_turns))
        # Populate the Area_lists array with correct values
        for p_area in p_areas:
            turn = int(p_area[0])
            player = p_area[1]
            players_area_list = p_area[2].replace("'", "").split(',')
            for area in players_area_list:
                if len(area.strip()) > 0:  # Don't include empty areas
                    Area_lists[AREA_INDEX[area.strip()], player_index[player],
turn] = 1
        # State of the board after each turn
        board_state_pattern = re.compile("([0-9]+), 'State of the Board', '(P;[A-Z
;a-z;_]+)', (\"|\')({.*})(\"|\')")
        states = re.findall(board_state_pattern, file)
        # Initialize empty unit list arrays
        Unit_lists = np.zeros((42, num_players, total_turns))
        territory_forces = re.compile("\'([a-zA-z ]+)\': ([0-9]+)")
        # Gather the unit lists at every turn of the game.
        for state in states:
            turn = int(state[0])
            player = state[1]
            forces = re.findall(territory_forces, state[3])
            for territory, troop_count in forces:
                Unit_lists[T_INDEX[territory], player_index[player], turn] = int(
troop_count)
        # Turn the Unit list into something DataFrame Friendly
```

```
        df_Unit_list = np.zeros((total_turns, 42 * num_players))
        for turn in range(total_turns):
            df_Unit_list[turn, :] = Unit_lists[:, :, turn].T.reshape(42 *
    num_players)
        # Get the headers for the dataframe
        header = []
        for player in p_name_list:
            for territory in T_INDEX.keys():
                header.append('Player ' + str(player_index[player]) + ' ' +
    territory)
        # Create and populate the dataframe
        unit_df = pd.DataFrame(df_Unit_list)
        unit_df.columns = header
        # Add some new columns to the DataFrame
        for player in range(num_players):
            unit_df[f'Player {player} total territories'] = np.sum(Unit_lists[:,
    player, :] != 0, axis=0)
        # Add area control columns
        for player in p_name_list:
            p_index = player_index[player]
            for area in AREA_INDEX.keys():
                col_title = 'Player ' + str(p_index) + ' ' + area
                unit_df[col_title] = Area_lists[AREA_INDEX[area], p_index, :]
        #create some features
        for i in range(num_players):
            #get the columns for the continental control for that player
            x = list_player_continents(i)
            # Assuming that the following ordering, and rewards per continent
            #order = [North America,South America, Africa, Europe, Asia, Australia
    ]
            #rewards = [5,2,3,5,7,2]
            # then matrix multiplication gives the continental rewards
            unit_df[f'Player {i} Continental Reward'] = unit_df[x].values @
    [5,2,3,5,7,2]

            #get number of troops per player and country count
            # then get total troop increase per player per turn
            x = list_player_countries(player_num=i)
            unit_df[f'Player {i} Troop Count'] = unit_df[x].sum(axis=1)
            unit_df[f'Player {i} Country Count'] = (unit_df[x] > 0).sum(axis=1)
            unit_df[f'Player {i} Troop Increase Due to Country Count'] = unit_df[f
    'Player {i} Country Count'].apply(troop_income_due_to_country_possesion)
            unit_df[f'Player {i} Total Reinforcements'] = unit_df[f'Player {i}
    Troop Increase Due to Country Count'] + unit_df[f'Player {i} Continental Reward
    ']
        if debug:
            print(filename,unit_df.shape)
        # if new features are added, then these should be changed
        graph_features = ['player_cut_edges'
            , 'player_number_boundary_nodes'
            , 'player_boundary_fortifications'
            , 'player_average_boundary_fortifications'
            , 'player_connected_components']
        x = get_board_names_for_players(num_players)
        results = []
        for t in range(unit_df.shape[0]):
            l = get_graph_features(list(unit_df[x].iloc[t].values))
            results.append(l)
        r = np.array(results)
```

```
        #this is just to make sure the reshaping is done correctly
        # unit_df['graph_features'] = [x for x in np.array(results)]
        #get names for new columns / features
        new_names = []
        for feature in graph_features:
            for i in range(num_players):
                new_names.append(f'Player {i} {feature}')
        #add the features
        new = pd.DataFrame(r.reshape(unit_df.shape[0],len(graph_features)*
num_players),columns=new_names)
        unit_df = unit_df.merge(new,how='inner',left_index=True,right_index=True)
        # Add winner column
        if winner == 'None':
            unit_df['winner'] = np.nan
            for place in ['Second', 'Third', 'Fourth', 'Fifth', 'Sixth'][:
num_players - 1]:
                unit_df[place] = np.nan
            for i in range(num_players): #add per player soft score
                unit_df[f'Player {i} soft score'] = num_players**-1
        else:
            unit_df['winner'] = player_index[winner]
            unit_df[f'Player {int(player_index[winner])} soft score'] = 1
            # Add Loser columns columns
            loser_pattern = re.compile("'elimination', .*, '(P;[A-Z;a-z;_]+)'")
            losers = re.findall(loser_pattern, file)
            for i, place in enumerate(['Second', 'Third', 'Fourth', 'Fifth', '
Sixth'][:num_players - 1]):
                unit_df[place] = int(player_index[losers[-1]])
                unit_df[f'Player {int(player_index[losers[-1]])} soft score'] = (i
+2)**-1
                losers.pop()
        # Add total_turns column to the data set
        unit_df['total_turns'] = total_turns
        # Trim back the DataFrame to not include setup
        unit_df = unit_df.iloc[25*num_players - 1:]
        unit_df.index = np.arange(1, len(unit_df.index) + 1)
        # Add Proportion of borders territories
        for i in range(num_players):
            unit_df[f'Player {i} boundary territory %'] = unit_df[f'Player {i}
player_number_boundary_nodes']/unit_df[f'Player {i} total territories']
            unit_df[f'Player {i} boundary troop %'] = unit_df[f'Player {i}
player_boundary_fortifications']/unit_df[f'Player {i} Troop Count']
            unit_df[f'Player {i} boundary territory %'].fillna(-1)
            unit_df[f'Player {i} boundary troop %'].fillna(-1)
        # Save the dataframe to the hdf file.
        unit_df.to_hdf(output_dir + "/" + output_file + str(k) + '.hdf', '
dataframe')
        # Dictionary of other data that we may care about (other features)
        data = {"players": [(np.string_(p), player_index[p]) for p in p_name_list
]}  # HDF5 is picky about strings
        # Save as an HDF
        with h5py.File(output_dir + "/" + output_file + str(k) + '.hdf', 'a') as
hf:
            for k in data.keys():
                hf[k] = data[k]
```