# CPTS 315: Course Project

Elijah Stoll

December 13, 2022

## 1 Introduction

Battle Dice[2] is a board game developed by Aaron Gordon. It consists of 30 6-sided dice, each of which has 6 of 15 different abilities. These different abilities do different things: dealing damage, healing, or other effects to the player or their opponents. With a total of 4,060 different possible teams of 3 dice, one is naturally drawn to ask statistical questions related to the game. What team performs the best, i.e wins the most matches? What team performs the worst? What traits to the best performing set of teams have in common? Are there teams that do particularly well against the best team?

Battle Dice interested me as a topic of data analysis for a few reasons. It is a very fun game, and I wanted to have an concrete set of data to solidify some basic intuitions I and friends had about the strategy of the game. It has been a passion project of mine to create a simulator to allow for this concrete data set to be created. In the past I created a simulator in Python. I found that the slow speeds of Python hindered my ability to create a full set of these results. I wanted to rewrite it in a much faster language such as C#. The change in language did not magically solve all of my problems however, and I ended up having to go back to an earlier idea that I started with in the Python simulator: randomly choosing match-ups instead of iterating through all of them.

While this was disappointing, it gave me an opportunity to incorporate some of the things I learned in this class. Machine learning is ideal for situations where you want to extrapolate and learn something from incomplete data. In this case, I decided to simplify the questions I wanted to answer, limit the extent of my data mining task, and use machine learning to answer a more focused question: Which dice are the best to choose from the pool of 30 dice?

## 2 Data Mining Task

While having a complete set of data would be beneficial, there is simply far too much to generate in any reasonable amount of time. As previously mentioned, Battle Dice consists of 30 dice, which creates 4,060 possible teams of 3 dice. The total number of possible matches between two teams is 11,875,500, and this is even accounting for the fact that some team have overlapping sets of dice–we exclude those since in real life it is not possible for two players to have any of the same dice.

While I believed that writing the simulator that would create this data in C# would be fast enough to create my data-set, I found that this was not so. I was looking at weeks of runtime to generate a complete data-set. Unacceptable in my own free time, let alone with a set deadline like that this class demands. However, it also gave me an opportunity to incorporate the tools learned in this class.

One problem I had to address was how to create the data set in a way that allowed for it to be used in a machine learning platform. What I ultimately decided on was to focus exclusively on how often a team won, using its average win rate as the output classifier for machine learning. This was a bit new for me as most of the algorithms we have studied in this class have focused on binary categorizations as output, not discrete outputs like I was working with.

The input data set is given as a CSV file, where the first three items are the name of the dice that make up the team. So one team example is "Fairy,Nymph,Mermaid". The fourth element in each line is the average win rate of the team. So "Fairy,Nymph,Mermaid,0.5" would mean that the Fairy, Nymph, and Mermaid team won exactly half of the matches it was in. It is worth noting that ties are very possible in Battle Dice, but to simplify the output these ties are treated as win for both of the teams. The output win rate can vary from 0 to 1.

## 3 Technical Approach

The simulator is responsible for generating a large data set that a machine learning algorithm can then be trained on. Instead of calculating every possible pairing of two teams, we can instead use random sampling, or a Monte Carlo simulation[1], to pair off random teams with each other. Given enough random matches, the win rate of each team should closely approximate the actual win rate.

But critically, this approximation requires a large number of matches to be simulated. If one match per pair of teams is simulated, all that is known is that one team is better than another. There is no comparable metric between two teams that did not fight each other. This is a problem even for reasonable numbers of simulations. The solution then is to look not at each team, but at the dice that makes up each team. While each individual team may only get to fight a few times, the 30 dice that make up all possible teams are used many times. Thus, we can look more generally at the set of dice used for each team and ask something like "how often does a team that contains dice X win?"

First, the simulation is run many times, randomly picking two teams of three dice to face off. These two times are run through 1,000 battles, and the number of wins, losses, and ties recorded. If the team has already fought other teams, their wins, losses, and ties are added to their overall score. The resulting data-set after all computation is done consists of 4,060 teams and their average win rate.

The second step is to develop a model which predicts the performance of a set of dice based on this data. Because there are 30 dice, the inclusion or absence of any given dice is a feature in a vector of 30 unknown variables, where each variable can be considered the weight or the performance any given dice adds to a set, with some output $\hat{y}_i$, which represents the win rate of team $i$.

We will use various regressive models from Scikit-learn and evaluate their predictive scores over 10 iterations. In particular, we tried ordinary linear regression[3], ridge regression[4], stochastic gradient descent[7], support vector machine regression[8], and multi-layer perceptron regression[5]. To analyze our results we trained each model 10 times over the shuffled data-set, reserving 10% of the data for testing and scoring. Python code for this process is shown in the appendix.

## 4 Evaluation Methodology

The data-set that we will be training on is the previously generated win rate of each possible team of dice, as mentioned in the previous section. We will evaluate the performance of each model by its score. For all of the models we will be using, this score is the coefficient of determination $R^2$[6]. This is different than a classifier algorithm because output scores are not binary, where the score is simply the number of accurate predictions divided by the total number of predictions.

The best possible score is 1, and the score can also be negative. We will take the best model that we find and perform further analysis on it, comparing the predicted outputs and the weight that each individual dice gives to that win rate to the initial data set, seeing what inuitive conclusions can be drawn, and if the predictions given makes sense from the initial data set.

## 5 Results and Discussion

After testing the five models on the data set, we found that linear, ridge, and MLP regression all performed equally well. Of the three, MLP had the greatest variance between runs, achieving as high as a score of 0.733 and as low as 0.1978. Linear and ridge achieved much more consistent scores, and in fact their scores were almost identical. SGD performed fourth, with a wide margin that does reach into the range of linear and ridge models, but overall performance was worse. Finally, SVM completely failed in comparison to all other models, with a consistently negative score. See figure 1 and table 2. We decided on the ridge regression model to continue our analysis.

The next step was to predict the win rate of only a single dice. These predicted win rates can be thought of as the "weight" of having a certain dice on a team. The single best dice one could have in their team is the Nymph dice, with a win rate of 0.6979, followed by Zombie and Shadow. A full list of these predicted win rates can be seen in table 3. Interestingly enough, the team Nymph, Shadow, Zombie itself is ranked 68th from the data itself. The highest recorded win rate from the data is Nymph, Alien, Goblin, using the 1st, 5th, and 15th highest ranked dice. In fact, we can compare the predicted win rates with how often each dice appears in the top quartile, see figure 4. At worst, the predicted win rate is off by 6 places in the case of Zombie, Cyclops, Mermaid, and Hero, but more often than not the predicted win rate is within 0-2 places of the dice's top quartile count.

We find that linear regression was an accurate and consistent predictor of how well a dice did in practice. The score was limited to about 0.6 (using $R^2$), but it achieved this score very reliably. Using MLP, a neural network, we were capable of getting much better scores, but this was unpredictable. Of the two linear regression models, we chose ridge because of the improvements it offers over ordinary least squares regression. Ridge imposes a penalty on the magnitude of its coefficients, encouraging smaller coefficients. This was certainly the case for us, the ordinary least squares model gave coefficients that were all almost exactly $-9.618 * 10^1 0$, while ridge gave us coefficients all between $-0.05$ and 0.05, for the same or oftentimes even better results.

## 6 Lessons Learned

In retrospect, there are lots of things I would have done different for this project. I think that I should have focused sooner on incorporating machine learning into the generation of real results. With that in mind, I would have changed how I modeled my data. I chose somewhat arbitrarily to focus my data-set on predicting a set of dice's overall win rate, but this is something that is easy to generate from the data-set: after all, all 4,060 teams are already represented in the output of the simulator.

What I would have done instead is to keep the original plan of tracking two teams. So my data set would look like team 1 dice, team two dice, and the average win rate of team 1 vs team 2. Then I would have used machine learning to fill in the blanks, letting it train on which of those two teams would win. It could then take an arbitrary set of dice for each team and predict which would win. This is similar, I imagine, to what we did in homework 3, question 4. It would change my continuous, regression-based analysis to a discrete, classifier based one.

That being said, I am not convinced that I could have generated enough data for an ML algorithm to train on. Three dice per team is not a lot of features per data point, and I am worried that even if I had changed my approach, there would have been far too many gaps to give adequate, accurate results.

## 7 Acknowledgements

Thanks to Aaron Gordon for creating this game.

## References

[1] IBM Cloud Education. *Monte Carlo Simulation.* Aug. 24, 2020. URL: `https://www.ibm.com/cloud/learn/monte-carlo-simulation` (visited on 12/11/2020).

[2] Aaron Gordon. URL: `https://www.finalspark.org/battle-dice` (visited on 12/10/2020).

[3] scikit-learn.org. *Linear Models.* URL: `https://scikit-learn.org/stable/modules/linear_model.html#ordinary-least-squares` (visited on 12/12/2020).

[4] scikit-learn.org. *Linear Models.* URL: `https://scikit-learn.org/stable/modules/linear_model.html#ridge-regression-and-classification` (visited on 12/12/2020).

[5] scikit-learn.org. *Neural network models (supervised).* URL: `https://scikit-learn.org/stable/modules/neural_networks_supervised.html#multi-layer-perceptron` (visited on 12/12/2020).

[6] scikit-learn.org. *sklearn.linear$_m$odel.LinearRegression.* URL: `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression.score` (visited on 12/12/2020).

[7] scikit-learn.org. *Stochastic Gradient Descent.* URL: `https://scikit-learn.org/stable/modules/sgd.html#regression` (visited on 12/12/2020).

[8] scikit-learn.org. *Support Vector Machines.* URL: `https://scikit-learn.org/stable/modules/svm.html#regression` (visited on 12/12/2020).

## 8 Appendix

```
import pandas as pd
from sklearn import linear_model
from sklearn.neural_network import MLPRegressor
from sklearn import svm

# Get the full set of features.
def getAllDice(filename: str):
    features = set()
    with open(filename, 'r') as f:
        for line in f.readlines():
            tokens = line.split(',')
            features.add(tokens[0])
    return features

# Loads the data from the CSV file and parses it.
def loadData(filename:str, featureSet: set):
```

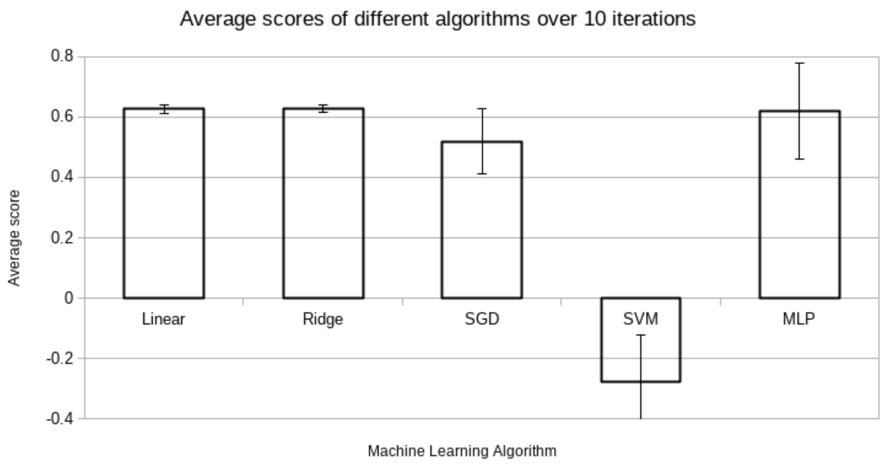

Figure 1: Average scores of different machine learning algorithms over 10 iterations on randomized data-set, with 10% of the data set reserved for testing each time. Error bars show standard deviation.

| Linear | Ridge | SGD | SVM | MLP |
|---|---|---|---|---|
| 0.6239 | 0.6245 | 0.4374 | -0.2948 | 0.5937 |
| 0.6368 | 0.6364 | 0.6062 | -0.1954 | 0.7115 |
| 0.6456 | 0.6447 | 0.5346 | -0.3288 | 0.7194 |
| 0.6263 | 0.6262 | 0.5045 | -0.4841 | 0.7099 |
| 0.6424 | 0.6421 | 0.6223 | -0.1757 | 0.7331 |
| 0.6090 | 0.6089 | 0.6010 | -0.1240 | 0.6856 |
| 0.6276 | 0.6278 | 0.4144 | -0.1688 | 0.5738 |
| 0.5969 | 0.6082 | 0.2939 | -0.5990 | 0.1978 |
| 0.6349 | 0.6347 | 0.5652 | -0.1793 | 0.6382 |
| 0.6158 | 0.6159 | 0.6094 | -0.2083 | 0.6259 |

Figure 2: Average scores of different machine learning algorithms over 10 iterations on randomized data-set, with 10% of the data set reserved for testing each time.

| Dice | Predicted Win Rate |
|---|---|
| Nymph | 0.6979 |
| Zombie | 0.6813 |
| Shadow | 0.6812 |
| Yeti | 0.6797 |
| Goblin | 0.6789 |
| Cyclops | 0.6765 |
| Fairy | 0.6761 |
| Mermaid | 0.6759 |
| Pirate | 0.6742 |
| Unicorn | 0.6737 |
| Cleric | 0.6689 |
| Puppet | 0.6687 |
| Hero | 0.6671 |
| Vampire | 0.6658 |
| Alien | 0.6646 |
| Troll | 0.6615 |
| Wizard | 0.6594 |
| Phoenix | 0.6583 |
| Dragon | 0.6570 |
| Werewolf | 0.6556 |
| Kraken | 0.6541 |
| Assassin | 0.6425 |
| Knight | 0.6516 |
| Mage | 0.6512 |
| Echnida | 0.6390 |
| Robot | 0.6370 |
| Elf | 0.6357 |
| Monk | 0.6332 |
| Giant | 0.6223 |
| Tank | 0.6112 |

Figure 3: Ridge model predictions for win rates of a team consisting of just one die.

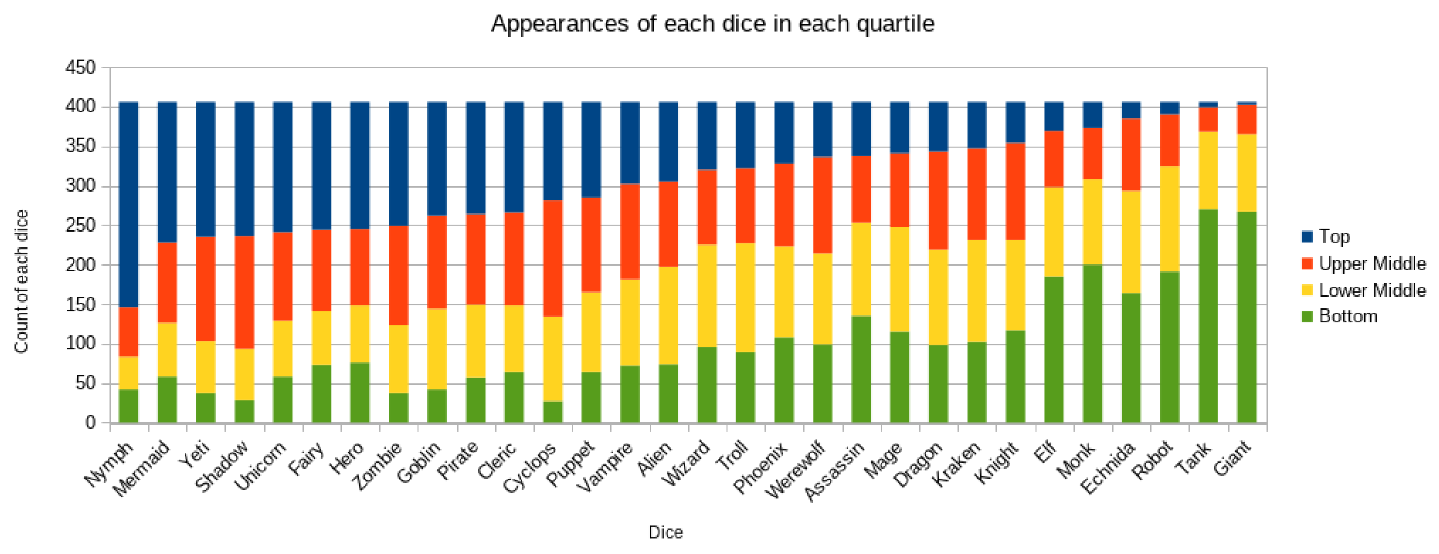

Figure 4: How often each dice appears in each quartile in the actual data set.

```
    results = pd.DataFrame(columns = list(featureSet) + [
        'Win_Rate'])
    with open(filename,'r') as f:
        for line in f.readlines():
            tokens = line.split(',')
            # Get first 3 tokens, each of the dice.
            row = dict(zip(featureSet,len(featureSet)
                *[0]))
            for die in tokens[:3]:
                row[die] = 1

            row['Win_Rate'] = float(tokens[-1])
            results.loc[len(results.index)] = row
    return results

if __name__ == "__main__":
    featureSet = getAllDice('dice.csv')
    data = loadData('results.csv',featureSet)
    resultScores = pd.DataFrame(columns = ['Linear', '
        Ridge', 'SGD','SVM','MLP'])

    # Run each model through 10 iterations.
    for i in range(0,10):
        # Shuffle the dataset
        data = data.sample(frac = 1)

        # Use 10% of data for tests.
        trainingData = data.loc[:3654, data.columns != '
            Win_Rate']
        trainingLabels = data.loc[:3654, data.columns ==
            'Win_Rate']
        testData = data.loc[3654:, data.columns != 'Win_
            Rate']
        testLabels = data.loc[3654:, data.columns == 'Win
            _Rate']

        model = linear_model.RidgeCV()
        model.fit(trainingData, trainingLabels)
        score = model.score(testData,testLabels)
        print("RidgeCV_score:_", score)

        model2 = linear_model.SGDRegressor()
        model2.fit(trainingData, trainingLabels.values.
            ravel())
        score2 = model2.score(testData, testLabels)
        print("SGDRegressor_score:_", score2)
    model3 = MLPRegressor(solver='lbfgs')
    model3.fit(trainingData, trainingLabels.values.
        ravel())
    score3 = model3.score(testData, testLabels)
    print("MLP Regressor: ", score3)

    model4 = svm.SVR()
    model4.fit(trainingData, trainingLabels.values.
        ravel())
    score4 = model4.score(testData, testLabels)
    print("Support Vector Regression: ", score4)

    model5 = linear_model.LinearRegression()
    model5.fit(trainingData, trainingLabels.values.
        ravel())
    score5 = model5.score(testData, testLabels)
    print("Linear Regression score: ", score5)

    scores = {'Linear':score5, 'Ridge':score, 'SGD':
        score2, 'SVM':score4, 'MLP':score3}
    resultScores.loc[i] = scores

print(resultScores)

# Evaluate each feature in ridge model. Used to get
    the score for a "single" die.
toPredict = pd.DataFrame(columns = list(featureSet))
for row in featureSet:
    r = dict(zip(featureSet, len(featureSet)*[0]))
    r[row] = 1
    toPredict.loc[len(toPredict.index)] = r

toPredict['Predicted Win Rate'] = model.predict(
    toPredict)
```