CS:GO Round Win Prediction using Supervised Learning

Sriram Boddeda (5033) Mark Cunningham (4033)

Abstract

This paper is concerned with testing a variety of supervised learning algorithms on a dataset containing round-by-round information about various conditions in the game of Counter-Strike: Global Offensive (CS:GO). We theorize that we will be able to outperform a random predictor using any of our algorithms to predict the winner of each round. We demonstrate our implemented algorithms from scratch and explain the different hyperparameters we used to obtain our results.

1. Domain

CS:GO is a competitive first-person shooter video game that was released in 2012. Despite being more than a decade old, it continues to gain popularity and just recently logged a new record of 1.8 million concurrent players on May 6, 2023. Oddly enough, the sequel to CS:GO was announced on the day of our presentation.

Two teams of five players face off against each other taking on the roles of the Terrorists (T) and Counter-Terrorists (CT). After 15 rounds have been played, the half concludes, and the teams' swap sides. Each round has a time limit of just under two minutes and players start with 100 health points. If they survived the previous round, they retain the equipment they survived with. The T-side must plant the bomb at either of the two bombsites located on the map and prevent the CT-side from defusing it. If the bomb is defused, the CT side automatically wins the round. Alternatively, the T-side can simply eliminate the CT side without planting the bomb and win the round. The CT side must either eliminate the T-side or successfully defuse the bomb if it was planted. If the bomb has not been planted after the time limit and there are still players alive on both sides, the CT side wins the round.

1.1. Economy

At the surface, CS:GO appears to be a simple FPS where the team who is better at clicking heads wins, and while this may be partially true, there are many other factors that make this game a tactical masterpiece. The personal balance of each player used to purchase equipment is one such example. At the start of each half, players start off with just the default pistol and $800. Players receive money by eliminating opponents, completing objectives, and winning or losing rounds. The T-side is awarded money for planting the bomb, while the CT-side is awarded money for defusing it. Losing a round grants much less than winning a round, but losing consecutive rounds will cause the team's loss bonus to increase. This is to prevent one side from completely snowballing the other.

Teams have to make decisions on how much to invest in a round, and these decisions can greatly affect subsequent rounds. For example, if one team goes into a round investing all the money they currently have (full buy) and loses, they will most likely not have enough money to fully buy the next round, unless they have the maximum loss bonus.

A common full buy round will typically have all players using a rifle, armor and helmet, and four pieces of utility. A half-buy scenario could include all players using an upgraded pistol, armor without a helmet, and a few scattered pieces of utility.

2. Related Work

In a paper by (Marshall et al., 2022) and colleagues, they showed how a long short-term memory (LSTM) neural network can outperform convolutional (CNN) and recurrent (RNN) neural network when predicting when a player would die in CS:GO in realtime. One of the most important features for their predictions was player equipment value, which also happens to be the feature that helped train and predict our models the most.

(Xenopoulos et al., 2020) and colleagues also sought to estimate win probability, but also used many more features than our paper does, including a unique graph-based distance measure that can find the distance of any player to any bombsite or other player. When comparing feature importance, they found equipment value to be the highest importance, followed by health remaining.

Predicting the outcomes of entire matches in CS:GO has been studied before in a paper by (Björklund et al., 2018). They used some of the same techniques for data preprocessing that we ended up using, such as feature normalization by subtracting the mean from the values and dividing the values

by the standard deviation. Skill rating and kill-death ratios proved to be the most important features when predicting the outcomes of online CSGO matches where each player has a skill rating.

A paper by (Rish, 2001) investigates the Naïve Bayes classifier algorithm's effectiveness on various datasets and compares it with other algorithms. Despite its simplistic assumptions, the algorithm can be highly effective in real-world classification tasks, particularly in text classification. The study provides valuable insights into the algorithm's practical applications and its strengths and limitations.

Predicting win outcomes based on the current game state is nothing unique to Counter-Strike, as (Yurko et al., 2019) used Random Forests to predict football game outcomes. They chose to focus on "offensive skill positions" like quarterback and runningback the various states associated with those positions.

3. Hypotheses

1. The Perceptron model will perform better than a random classifier.
2. The Logistic Regression model will perform better than a random classifier.
3. The Random Forest model will perform better than a random classifier.
4. The Naive Bayes model will perform better than a random classifier.

4. Experiments

4.1. Dataset

As mentioned in the proposal, the economy of each team going into a round and subsequently the equipment value that each player can purchase can greatly affect the ability of a team to win a given round. Creating a custom dataset ended up appearing like it would take too much time, so we instead settled for the Kaggle dataset mentioned in our presentation. We used attributes from 'economy.csv' and 'results.csv'. The economy file shows the equipment value for each team after they have purchased in a round. The results file shows more broad data like the team names and their world ranking.

We chose to narrow our scope from predicting round wins and match wins to just round wins. Since we're doing round-by-round predictions, we needed to transform the dataset. We chose to make every round in the dataset an individual row. The features are the current round number, team name, side of the map the team is on (T or CT), team 1 equipment value, team 2 equipment value, team 1 ranking, team 2 ranking, and the map that the round was played on.

Mark implemented the Perceptron and Sriram chose to do Logistic Regression. We first split our data into an 80% train set and a 20% validation set. We then took the 80% and divided it into a 75% train set and a 25% test set.

4.2. Perceptron

Implementing the perceptron began with identifying the weight vector, which appeared to be a vector of length five that contains the features described earlier. However, the map that the teams played on is a categorical attribute. Simply changing the string values into numbers wouldn't work since the perceptron is a linear model that uses a hyperplane to divide the decision space. Instead, I used pandas' get_dummies function, which turned the single map label into 10 columns since there are 10 total maps in the dataset. This way, each map can be considered separately from the rest since only one will have a value that isn't zero when multiplying the weight vector. I used a sigmoid function as the activation function, and mean squared error as the loss function.

The ROC curve is shown in Figure 1. The algorithm was able to achieve much better results than a baseline random classifier using money, rank, and the map. I also tested the perceptron using only one feature at a time. The resulting prediction accuracy for each feature is listed in Table 1. As expected, money (equipment value) gives the greatest amount of information gain among the three individual features. Surprisingly, rank by itself performs worse than a random classifier. However, combining all three features resulted in the greatest accuracy of 67%.

Features	Accuracy
Rank	0.47
Maps	0.51
Money	0.63
Rank, Maps, Money	0.67

Table 1. The prediction accuracy when including various features.

Another interesting note is that when the map isn't included as a feature to create the model, the ROC curve is much wavier. This suggests that having the extra information about which map is being played allows the perception to make a more informed and uniform decision. I believe this is because some maps are inherently easier to win during a "force-buy" round. A force-buy round is when a team that doesn't have enough money to fully purchase all standard equipment decides to purchase the best equipment they can instead of saving money for the next round. Maps in which a force-buy would be easier on could have attributes like closer angles, which can level the playing field against an opponent with a more expensive, long-range rifle.

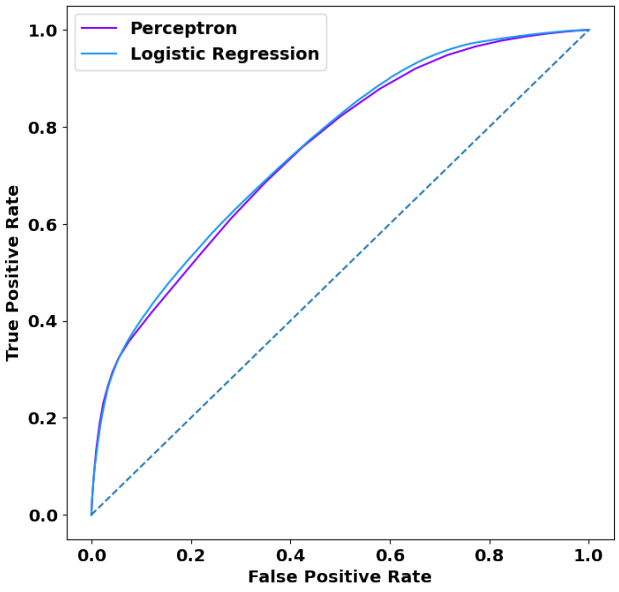

Figure 1. The ROC Curve for both the Perceptron and Logistic Regression algorithms.

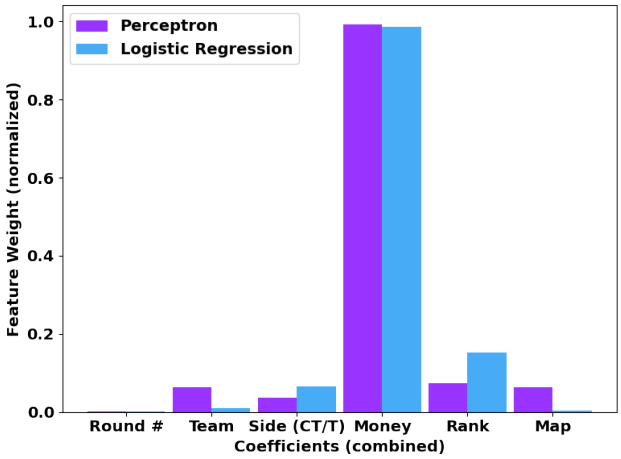

Figure 2. The weights for both the Perceptron and Logistic Regression algorithms.

Features	Accuracy
Rank	0.54
Maps	0.51
Money	0.67
Rank, Maps, Money	0.67

Table 2. The prediction accuracy when including various features.

4.3. Logistic Regression

The first thing to check while implementing a Logistic regression model is to check if all the input variables are numeric. They can be either continuous or categorical but have to be numerical in nature. We converted all the string values to numerical values and which map to categorical label variables. Next, I ensured that the input variables are independent of one another since logistic regression assumes that there is no multicollinearity among the independent variables. This can be checked by examining the correlation matrix between the independent variables, and if there is a high correlation between two or more variables, one of them may need to be removed from the model to avoid multicollinearity. Additionally, the input variables should be normalized to ensure that they are on the same scale and to help the model converge faster during training.

The ROC curve for the Logistic Regression model is shown beside the perceptron algorithm which is shown in Figure 1. The model was able to get 67% accuracy including the following features: money before each round, each team's rank, and a map of the game. The accuracy with each combination of features is shown in Table 2. For some reason, we're getting the same accuracy with or without ranks, and maps. But we're getting lower accuracy without the data of money each team has.

For our next two algorithms, Mark chose to implement the Random Forest algorithm and Sriram chose Naive Bayes.

4.4. Random Forest

A random forest is a collection of decision trees created from a randomly sampled smaller subset of features. To create my implementation, I first created the RandomForest class. The fit method creates the decision trees based on a number of specified parameters including the number of estimators, number of features, max depth, minimum sample split, and minimum samples per leaf. Finding the right value for each of these parameters proved to be a challenge since there is a somewhat delicate balance to strike that defines how under-fitted or over-fitted the model will be.

Each decision tree uses a bootstrapped subset of the input training set with replacement so the forest is exposed to the dataset in a distributed manner. I structured each decision tree to contain a collection of nodes. Each tree is fit recursively by finding the best split among the feature values at each node. The best split is found using Gini impurity gain, which calculates how effective the feature was at splitting the values. This is achieved by calculating the difference between the Gini impurity of the parent node and the weighted average of the Gini impurity of the two child nodes.

Once the model is trained, all trees created by the model are used to generate predictions by averaging their binary predictions and rounding them up or down. Combining the predictions of independently generated trees can help

to lessen the effects of over-fitting that may have occurred during model fitting. To keep track of how the model was performing, I created several variables that are stored within each node in each tree which include the obvious choices like the child nodes and whether or not the node is a leaf node. I also included the probability distribution of the classes for the samples that end up in a leaf node. If the distribution at a leaf node isn't 0:1 or 1:0, that means that there is uncertainty in the prediction.

Despite creating a class more than triple the lines of the Perceptron, the Random Forest performed very poorly. The general algorithm works fine, but there must be a mistake somewhere in my implementation. It successfully creates the trees, makes predictions, and chooses the most important features to place at the root of the tree, but still outputs weak results.

4.5. Naive Bayes

Naive Bayes is a popular machine learning algorithm for classification tasks that is based on Bayes' theorem. The algorithm is called "naive" because it assumes that the features are independent of each other, which simplifies the calculations. In practice, this assumption is often violated, but Naive Bayes can still perform well in many applications.

To create the logic for the Naive Bayes model, we first compute the prior probabilities of each class label and the mean and variance of each feature for each class label using a training dataset. Then, for a given test sample, we compute the likelihood of each feature vector given each class label using a Gaussian probability density function (for continuous features) or a multinomial distribution (for discrete features). We then use Bayes' theorem to compute the probability of each class label given the feature vector. The predicted class label is the one with the highest probability.

To implement this logic in code, we can create a NaiveBayes class with three main methods: 'fit()', 'predict()', and 'gaussian_pd()'. The 'fit()' method takes a training dataset as input and computes the prior probabilities and mean and variance of each feature for each class label. The 'predict()' method takes a test dataset as input and computes the probability of each class label given the feature vector using the 'gaussian_pd()' method. The predicted class label is the one with the highest probability. The 'gaussian_pd()' method computes the value of the Gaussian probability density function for each feature.

After training the model, it correctly predicted around 66.6% of the test samples which is similar to the perceptron and logistic regression models.

Overall, Naive Bayes is a simple and fast algorithm that can perform well on many classification tasks, especially when the number of features is large compared to the number of training samples. The implementation of the Naive Bayes model logic involves understanding the underlying mathematical concepts, creating a class-based approach to implementing the algorithm and testing the implementation to ensure that it works correctly.

5. Summary and Future Work

In summary, we are able to predict round-win outcomes better than a baseline random predictor in a majority of our models. The Perceptron, Logistic Regression, and Naive Bayes models all performed well and had extremely similar characteristics. Even though they were implemented completely differently, they still separated the data nearly identically. This is most likely because our dataset is linearly separable and convergence will occur regardless of the model. We thought that adding in the round number might have some impact on the models, but as shown in Figure 2, their weighting is hardly even visible. Including which side of the map each team was on seemed to aid all models, even though they are all still dominated by equipment value.

For starters, improving or fixing the random forest model would be our highest priority. Future work that could be done could include extending our implementation to a live update system that tracks win probability during a live match.

References

Björklund, A., Visuri, W. J., Lindevall, F. & Svensson, P. (2018). Predicting the outcome of cs:go games using machine learning.

Marshall, S., Mavromoustakos-Blom, P. & Spronck, P. (2022). Enabling real-time prediction of in-game deaths through telemetry in counter-strike: Global offensive. In *Proceedings of the 17th International Conference on the Foundations of Digital Games*, FDG '22. New York, NY, USA: Association for Computing Machinery.

Rish, I. (2001). An empirical study of the naive bayes classifier. *IJCAI 2001 Work Empir Methods Artif Intell*, *3*.

Xenopoulos, P., Doraiswamy, H. & Silva, C. T. (2020). Valuing player actions in counter-strike: Global offensive. *CoRR*, *abs/2011.01324*.

Yurko, R., Ventura, S. & Horowitz, M. (2019). nflwar: a reproducible method for offensive player evaluation in football. *Journal of Quantitative Analysis in Sports*, *15(3)*, 163–183.