Highlights

Comparative Research on Predictive Models Based on MOBA Game Data Set

Yumin Xu,Michael Vigil,Logan Decker

- Compare the performance of the same training model on different data sets.
- Compare the performance of different training models on the same data set.

Comparative Research on Predictive Models Based on MOBA Game Data Set

Yumin Xu[a,c,*,1], Michael Vigil[b,d] and Logan Decker[b,c,2]

[a]New Mexico Tech, 801 Leroy Place, Socorro, 87801 NM,

ARTICLE INFO

Keywords:
Decesion Tree
K-NN
Naive Bayes
Win-rate prediction

ABSTRACT

With the rapid popularity and explosive development of MOBA e-Sports (Multiplayer Online Battle Arena electronic sports), much research is devoted to automatically predicting game results. Although these studies have great potential in various applications, previous studies are either based on prior features such as the historical win rate of game players or based on real-time features during the game. So we want to ask how much the performance of different algorithms varies for the same feature type of dataset. If we use the same algorithm to process different types of datasets, does the difference between the prior and the posterior lead to the difference in model performance? In this paper, three common machine learning algorithms, decision trees, K-NN, and Naive Bayes, are chosen to model two different datasets. By testing the performance of the models, we find that the impact of the characteristics of the datasets on the model performance sometimes exceeds the impact of the different algorithms on the model performance.

1. Introduction

For a comparative study, we found two data sets from the data set open source website in the same domain but with different feature types. They are the LOL dataset and the DOTA data set, two MOBA games that are popular all over the world today, and they have slightly different rules but the same basic gameplay. However, the two data sets are selected with different features.
The DOTA data set is reasonably sparse as only 10 of 113 possible roles are chosen in a given game. All games were played in a space of 2 hours on the 13th of August, 2016. Each row of the data set is a single game with the following features (in the order in the vector): 1. Team won the game (1 or -1) 2. Cluster ID (related to location) 3. Game mode (eg All Pick) 4. Game type (eg. Ranked) 5 - end: Each element is an indicator for a hero. Value of 1 indicates that a player from team '1' played as that hero and '-1' for the other team. Hero can be selected by only one player each game. This means that each row has five '1' and five '-1' values. This dataset contains the first 10min. stats of approx. 10k ranked games (SOLO QUEUE) from a high ELO (DIAMOND I to MASTER). Players have roughly the same level.
For LOL dataset, each game is unique. The gameId can help us to fetch more attributes from the Riot API. There are 19 features per team (38 in total) collected after 10min in-game. This includes kills, deaths, gold, experience, level. The column blueWins is the target value (the value we are trying to predict). A value of 1 means the blue team has won. 0 otherwise. By the way, the two datasets have no missing value.
Our study is divided into three main steps. Step one is Training. In training step, we apply the Decision Tree model on the two data sets respectively then apply the K-NN model on the two data sets respectively. Finally, we apply the Naive Bayes model on the two data sets respectively. Step two is evaluating. In evalutating step, based on the first step, we can get six training models. Use ROC curve and Confusion Matrix to evaluate these six models and analyze their pros and cons. Last Step is comparing. In the last step, we will have two types comparison. One is Horizontal comparison: Compare the performance of the same training model on different data sets. Another one is Longitudinal comparison: Compare the performance of different training models on the same data set. All the above experiments will be performed in the Rstudio environment.

2. Training Model

2.1. Decision Tree

CART algorithm (Classification And Regression Tree, classification and regression tree) is a kind of decision tree, proposed by Leo Breiman, Jerome Friedman, Richard Olshen and Charles Stone in 1984. It can be used for both

ORCID(s):

classification and regression. This section will mainly introduce the R language implementation of the CART algorithm for classification.

2.1.1. LOL Dataset

1. Read Data

Before read data, we load all libraries we will use. Because the original variable name is too long, we use names command to rename them. We use as.factor command transformed the dependent variable into a factorial format.

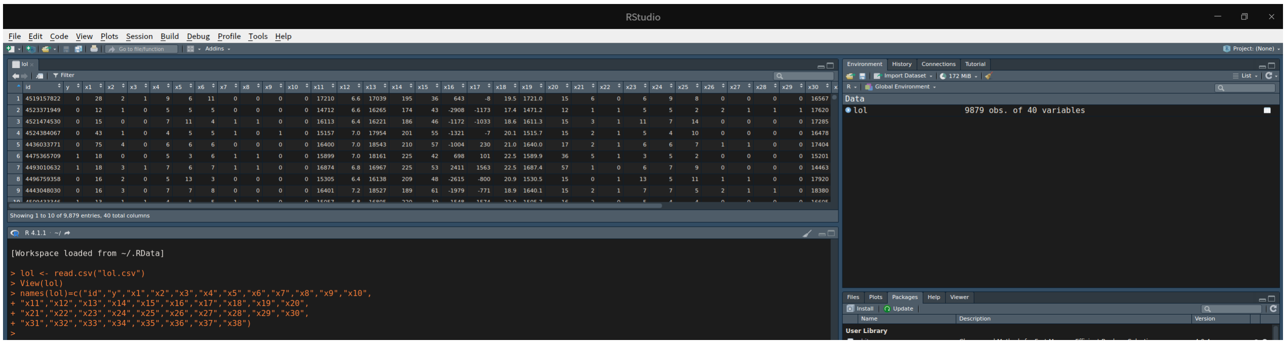

Figure 1:

2. Create training set and test set

We randomly select 70% of the data as the training set, named train, and the remaining data as the test set, named test.

```
> lol$y = as.factor(lol$y)
> select<-sample(1:nrow(lol),length(lol$id)*0.7)
> train=lol[select,]
> test=lol[-select,]
```

Figure 2:

3.Build a CART model

We use tc<-rpart.control(minsplit = 50,minbucket = 20,maxdepth = 30,xval =10,cp = 0.001) to set the conditions for Pre-pruning. Among them, minsplit is the minimum number of branch nodes, which means that if it is greater than or equal to 50, then the node will continue to divide, otherwise it will stop. minbucket is Minimum sample number of leaf nodes. maxdepth is Tree depth. xval is Number of cross-validation xval=10 is 10-fold cross-validation (the dataset is divided into 10 groups and fitted 10 times, with the ith fit trained with data other than the ith group and the ith group used for prediction; the aim is to reduce misclassification). The full name of cp is complexity parameter, which refers to the complexity of a certain point, for each step of splitting, the model must improve the goodness of fit, used to save the unnecessary time wasted on pruning.

```
> library(rpart)
> library(rpart.plot)
> tc<-rpart.control(minsplit = 50,minbucket = 20,maxdepth = 30,xval =10,cp = 0.001)
```

Figure 3:

Then we use formular command and rpart.mod command to build a decision tree model. After the model is built we need to perform the post-pruning process. There are many post-pruning methods that can be used in categorical regression trees, such as: cost complexity pruning, minimum error pruning, pessimistic error pruning, etc. Here we only use the cost complexity pruning method.

```
> formular=y~x1+x2+x3+x4+x5+x6+x7+x8+x9+x10+x11+x12+x13+x14+x15+x16+x17+x18+x19+x20+x21+x22+x23+x24+x25+x26+x27+x28+x29+x30+
+ x31+x32+x33+x34+x35+x36+x37+x38
> rpart.mod=rpart(formular,data = train,method = "class",
+ parms = list(prior=c(0.6,0.4),loss=matrix(c(0,1,2,0),nrow=2),split="gini"),
+ control = tc)
```

Figure 4:

The rpart package provides a pruning method for complexity loss pruning. rpart.mod$cp will tell us how much cp and what is the average relative error when the model is split to each layer.

```
> rpart.mod$cp
            CP nsplit rel error   xerror       xstd
1  0.095707409      0 1.0000000 2.000000 0.02407540
2  0.014495293      2 0.8085852 1.367510 0.02255053
3  0.004936469      3 0.7940899 1.337816 0.02239696
4  0.004928857      7 0.7726163 1.309419 0.02223251
5  0.002907491      8 0.7676874 1.305071 0.02220433
6  0.002757252      9 0.7647799 1.306662 0.02221012
7  0.002305615     16 0.7417044 1.308094 0.02219101
8  0.002176666     18 0.7370932 1.316055 0.02222535
9  0.002028392     20 0.7327399 1.337486 0.02233063
10 0.002019023     24 0.7246210 1.336762 0.02232609
11 0.001740630     27 0.7185640 1.329508 0.02226915
12 0.001454624     29 0.7150827 1.332531 0.02225539
13 0.001434129     30 0.7136281 1.326155 0.02221614
14 0.001344649     35 0.7061357 1.334538 0.02223290
15 0.001315573     38 0.7021018 1.335842 0.02223998
16 0.001045432     43 0.6919440 1.339313 0.02224929
17 0.001007169     58 0.6713583 1.333949 0.02221330
18 0.001000000     60 0.6693439 1.333949 0.02221330
>
```

Figure 5:

The estimated error (xerror), standard error (xstd), and average relative error (xerror±xstd) of the cross-validation can also be printed as line graphs via "plotcp".

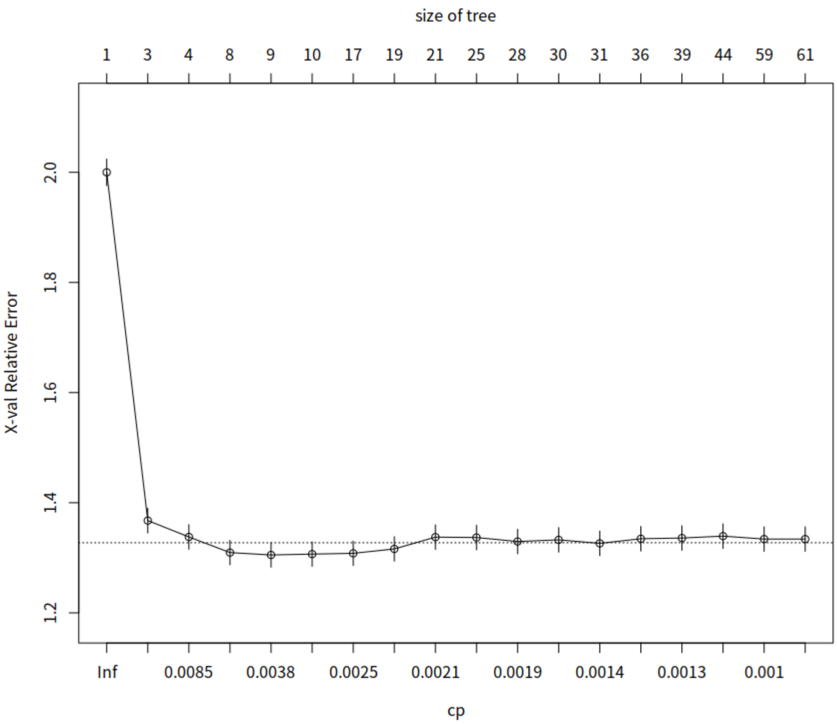

Figure 6:

The condition that the complexity pruning method satisfies is that when the estimated error (xerror) of the cross-validation is as small as possible (not necessarily the minimum value, but within one standard deviation (xstd) of the allowable minimum error), choose as large as possible The cp value. In our model we choose the method of cp with the smallest xerror.

```
> plotcp(rpart.mod)
> rpart.mod.pru<-prune(rpart.mod,cp=rpart.mod$cptable[which.min(rpart.mod$cptable[,"xerror"])])
> rpart.plot(rpart.mod.pru,branch=1,extra = 102,under = TRUE,faclen = 0,cex = 0.7,main="CART")
```

Figure 7:

We use rpart.plot() function to draw the final decision tree.

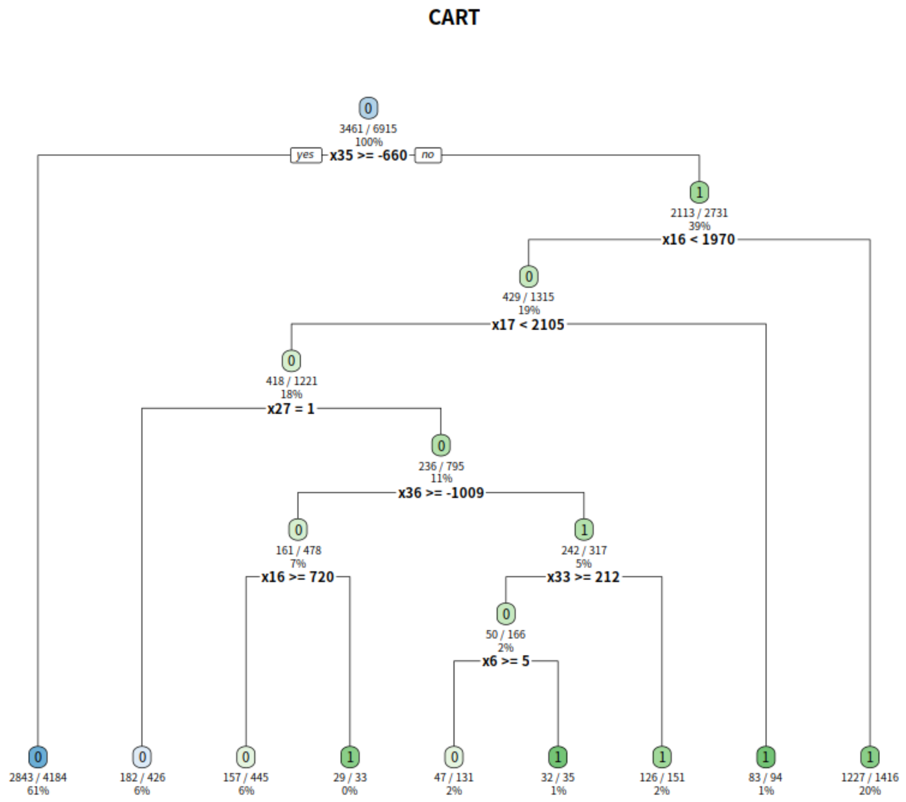

Figure 8:

At this point, the decision tree model based on the LOL dataset has been built.

2.1.2. DOTA Dataset

1. Read Data

Same as the LOL dataset, we load all libraries we will use at the beginning. We start by making two changes to the dataset. First, we use the command to modify the original dataset since there is no id column in the original dataset. This is done to facilitate subsequent modeling operations. Second, to avoid potential effects, we unified the symbols indicating wins and losses with the command. Making the form of representing wins and losses the same as the LOL dataset is to make the two datasets comparable to the maximum extent possible.

2. Create training set and test set

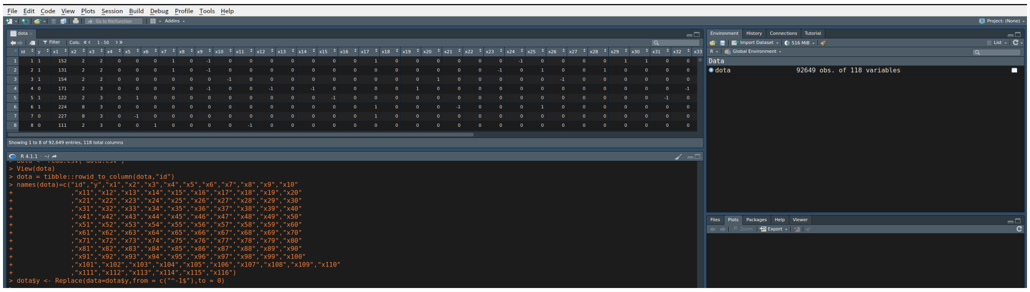

Figure 9:

We randomly select 70% of the data as the training set, named train, and the remaining data as the test set, named test.

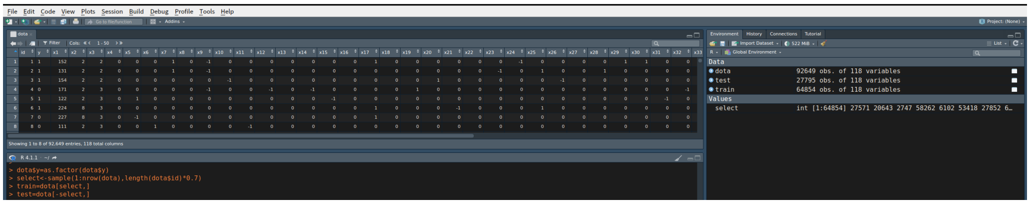

Figure 10:

3.Build a CART model

The number of variables is particularly large and because all of them may affect the final victory or defeat. Therefore we cannot reduce the number of variables at will. But the consequence of doing so is that the prepruning condition becomes very demanding.

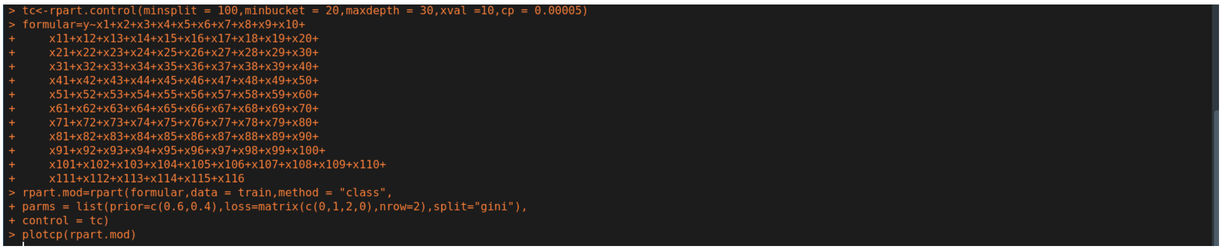

Figure 11:

Then we use formular command and rpart.mod command to build a decision tree model. After the model is built we need to perform the post-pruning process. There are many post-pruning methods that can be used in categorical regression trees, such as: cost complexity pruning, minimum error pruning, pessimistic error pruning, etc. Here we only use the cost complexity pruning method.

The rpart package provides a pruning method for complexity loss pruning. rpart.mod$cp will tell us how much cp and what is the average relative error when the model is split to each layer. The estimated error (xerror), standard error (xstd), and average relative error (xerror±xstd) of the cross-validation can also be printed as line graphs via "plotcp".

The condition that the complexity pruning method satisfies is that when the estimated error (xerror) of the cross-validation is as small as possible (not necessarily the minimum value, but within one standard deviation (xstd) of the allowable minimum error), choose as large as possible The cp value. In our model we choose the method of cp with

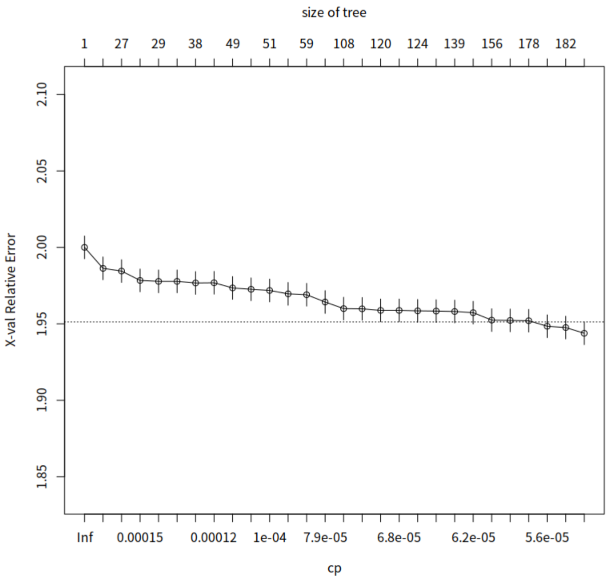

Figure 12:

```
> rpart.mod.pru<-prune(rpart.mod,cp=rpart.mod$cptable[which.min(rpart.mod$cptable[,"xerror"]),])
> rpart.plot(rpart.mod.pru,branch=1,extra = 102,under = TRUE,faclen = 0,cex = 0.7,main="CART")
```

Figure 13:

the smallest xerror.

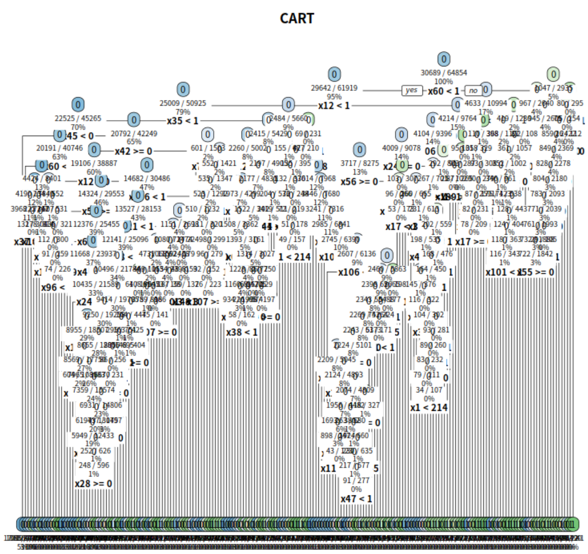

Figure 14:

We use rpart.plot() function to draw the final decision tree.
At this point, the decision tree model based on the DOTA dataset has been built.

2.2. K-NN

2.2.1. LOL Dataset

2.2.2. DOTA Dataset

2.3. Naive Bayes

2.3.1. Code

```r
#Setup
install.packages("naivebayes")
install.packages("MLmetrics")
library(naivebayes)
library(MLmetrics)

#Normalization Function
normalize <- function(x) {
    return ((x - min(x)) / (max(x) - min(x)))
}

#Read in data, combine DotA2 sets
lol = read.csv("high_diamond_ranked_10min.csv")
dota2_1 = read.csv("dota2Train.csv", header = FALSE)
dota2_2 = read.csv("dota2Test.csv", header = FALSE)
dota2 = rbind(dota2_1, dota2_2)

#Preprocess data
lol = lol[-1]
lol_normalized = as.data.frame(lapply(lol, normalize))
lol_normalized[1] = as.logical(lol_normalized$blueWins)
dota2[1] = replace(dota2[1], dota2[1] < 1, 0)
dota2_normalized = as.data.frame(lapply(dota2, normalize))
dota2_normalized[1] = as.logical(dota2_normalized$V1)

#Loop for training and testing
best_lol_f1 = 0
best_dota2_f1 = 0
for (val in 1:100) {
  #Split into train/test sets
    lol_train = sample(nrow(lol), replace = FALSE, 7903)
    lol.training = lol_normalized[lol_train,]
    lol.testing = lol_normalized[-lol_train,]
    dota2_train = sample(nrow(dota2), replace = FALSE, 82355)
    dota2.training = dota2_normalized[dota2_train,]
    dota2.testing = dota2_normalized[-dota2_train,]

  #Train the models
    lol_model = naive_bayes(lol.training[-1], lol.training$blueWins)
    dota2_model = naive_bayes(dota2.training[-1], dota2.training$V1)

  #Make predictions
    lol_pred = predict(lol_model, lol.testing[-1])
    dota2_pred = predict(dota2_model, dota2.testing[-1])

  #Calculate F1 scores
    lol_f1 = F1_Score(lol.testing$blueWins, lol_pred, positive = "TRUE")
    dota2_f1 = F1_Score(dota2.testing$V1, dota2_pred, positive = "TRUE")

```

Figure 15:

```r
  #Record model and data split for good performance
    if (lol_f1 > best_lol_f1) {
        best_lol_model = lol_model
        best_lol_train = lol.training
        best_lol_test = lol.testing
        best_lol_pred = lol_pred
        best_lol_f1 = lol_f1
    }
    if (dota2_f1 > best_dota2_f1) {
        best_dota2_model = dota2_model
        best_dota2_train = dota2.training
        best_dota2_test = dota2.testing
        best_dota2_pred = dota2_pred
        best_dota2_f1 = dota2_f1
    }
}

#LoL results
table(best_lol_test$blueWins, best_lol_pred)
F1_Score(best_lol_test$blueWins, best_lol_pred, positive = "TRUE")
Accuracy(best_lol_test$blueWins, best_lol_pred)
Precision(best_lol_test$blueWins, best_lol_pred, positive = "TRUE")
Recall(best_lol_test$blueWins, best_lol_pred, positive = "TRUE")

#DotA2 results
table(best_dota2_test$V1, best_dota2_pred)
F1_Score(best_dota2_test$V1, best_dota2_pred, positive = "TRUE")
Accuracy(best_dota2_test$V1, best_dota2_pred)
Precision(best_dota2_test$V1, best_dota2_pred, positive = "TRUE")
Recall(best_dota2_test$V1, best_dota2_pred, positive = "TRUE")

#Cross test
lol_cross_pred = predict(best_lol_model, best_dota2_test[-1])
dota2_cross_pred = predict(best_dota2_model, best_lol_test[-1])

#DotA2 model results on LoL data
table(best_lol_test$blueWins, dota2_cross_pred)
F1_Score(best_lol_test$blueWins, dota2_cross_pred, positive = "TRUE")
Accuracy(best_lol_test$blueWins, dota2_cross_pred)
Precision(best_lol_test$blueWins, dota2_cross_pred, positive = "TRUE")
Recall(best_lol_test$blueWins, dota2_cross_pred, positive = "TRUE")

#LoL model results on DotA2 data
table(best_dota2_test$V1, lol_cross_pred)
F1_Score(best_dota2_test$V1, lol_cross_pred, positive = "TRUE")
Accuracy(best_dota2_test$V1, lol_cross_pred)
Precision(best_dota2_test$V1, lol_cross_pred, positive = "TRUE")
Recall(best_dota2_test$V1, lol_cross_pred, positive = "TRUE")
```

Figure 16:

2.3.2. LOL Dataset

Training this data set was also generally quite easy. The League of Legends data set that we used had included Riot Game ID values for each of the data points, which we removed as a part of preprocessing. Other than that, the same conversion of ’0’ and ’1’ values to ’FALSE’ and ’TRUE’ values (respectively) for the ’Win’ category was performed, and the normalization to values between 0 and 1 for all non-win categories was also performed.
After this setup, the process was nearly identical to that performed while training the DotA2 set. For 100 iterations, the

data set was split randomly into 80-20 proportions where the larger portion was used for training the Naive Bayes model.

2.3.3. DOTA Dataset

Training this data set was also generally quite easy. The League of Legends data set that we used had included Riot Game ID values for each of the data points, which we removed as a part of preprocessing. Other than that, the same conversion of '0' and '1' values to 'FALSE' and 'TRUE' values (respectively) for the 'Win' category was performed, and the normalization to values between 0 and 1 for all non-win categories was also performed.
After this setup, the process was nearly identical to that performed while training the DotA2 set. For 100 iterations, the data set was split randomly into 80-20 proportions where the larger portion was used for training the Naive Bayes model.

3. Evaluating Model

3.1. Decision Tree

3.1.1. LOL Dataset

1.Importance of parameters

```
> rpart.mod.pru$variable.importance
      x16         x35         x17         x36         x11         x19         x27         x26          x6         x33         x37          x8         x13 
525.7835444 525.7835444 335.3620536 335.3620536 270.0017773 268.8928974   8.5282588   6.5663589   5.9950776   4.1661412   4.1661412   3.1230244   2.8828620 
       x4         x24         x32         x14         x18          x7         x23          x5         x12         x34 
  2.6036039   2.4634682   2.4034362   1.9707745   1.9707745   1.8017448   1.1036136   1.1036136   0.2469852   0.2202133 
```

Figure 17:

2. ROC curve

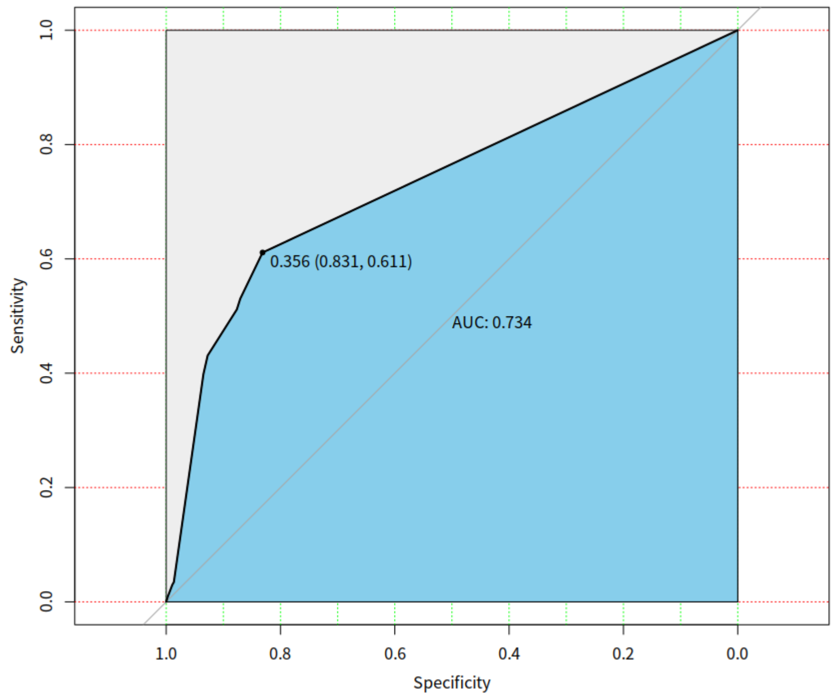

Figure 18:

3. Confusion matrix

```
> test$pre_p<-predict(rpart.mod.pru,test)[,2]
> test$pre=0
> test$pre[which(test$pre_p>0.356)]=1
```

Figure 19:

```
> print(cft <- table(test$pre,test$y))

       0    1
  0 1237  574
  1  251  902
> tp <- cft[2, 2]
> tn <- cft[1, 1]
> fp <- cft[2, 1]
> fn <- cft[1, 2]
> print(accuracy <- (tp + tn)/(tp + tn + fp + fn))
[1] 0.7216599
> print(sensitivity <- tp/(tp + fn))
[1] 0.6111111
> print(specificity <- tn/(tn + fp))
[1] 0.8313172
```

Figure 20:

Since the AUC = 0.734, the classifier obtained from our modeling outperforms random guesses. This classifier (model) can have some predictive value if the threshold value is properly set.
However, according to the confusion matrix, we found that this classifier is not sensitive enough, although it is able to filter out false positives better.

3.1.2. DOTA Dataset

1.Importance of parameters

```
> rpart.mod.pru$variable.importance
        x12         x60         x39        x106          x1         x56         x24        x109         x45         x35         x47         x21          x9         x42 
75.15594210 74.02760075 32.33811631 32.09994667 31.97746580 29.29544819 28.41990834 26.32861222 25.73339716 22.84241408 22.64418972 22.25100637 20.05337773 18.62485370 
        x77         x78         x17          x4         x25         x94          x5         x11         x29          x3          x8          x2         x75         x88 
16.79655319 15.24946816 14.46534416 14.19886542 14.12135640 13.65267967 13.49959588 12.80099934 12.50922887 12.27743077 11.13263708 11.07275392 10.80799795 10.60203432 
       x103         x96         x73         x28         x64         x62         x32        x107         x38         x22         x89        x114        x101          x7 
10.06400269 10.02166483  9.94885941  9.60787786  9.47625786  8.76404891  8.21698457  7.74322030  7.55413859  7.50957038  7.49057454  7.18154348  7.06697150  6.73694514 
        x44         x14         x16         x87         x10         x37        x115         x98        x112         x76        x113         x90         x51         x61 
 6.64952074  6.57382358  6.51101719  6.45953566  5.49719418  5.43219817  4.91845350  4.80867653  4.06612515  3.90665658  3.78875310  3.75732074  3.72317318  3.47209469 
        x59        x102         x65         x70         x99         x66         x55         x97         x18         x53         x49         x57         x48         x79 
 3.42769056  3.38335824  3.37023324  3.17466216  3.16381505  3.13451283  3.12375708  2.99303390  2.99181243  2.96598675  2.64408657  2.60397596  2.59626367  2.55202999 
        x91         x31        x110         x23          x6         x83         x81         x54         x58        x105         x15         x34         x26         x71 
 2.10240456  1.95385927  0.62904617  0.49799323  0.44961517  0.41457351  0.36766900  0.35777829  0.35725057  0.34981809  0.32894309  0.31300676  0.26511408  0.24914362 
        x68         x93        x108         x50         x72         x19         x63        x116         x40        x100         x36        x104         x92         x80 
 0.24122803  0.23535950  0.23445013  0.15672984  0.14861654  0.14441040  0.14399711  0.13872031  0.12840910  0.12687840  0.11503065  0.11024664  0.10158087  0.08563951 
        x86         x30         x95 
 0.08146356  0.05055813  0.01783682 
```

Figure 21:

2. ROC curve

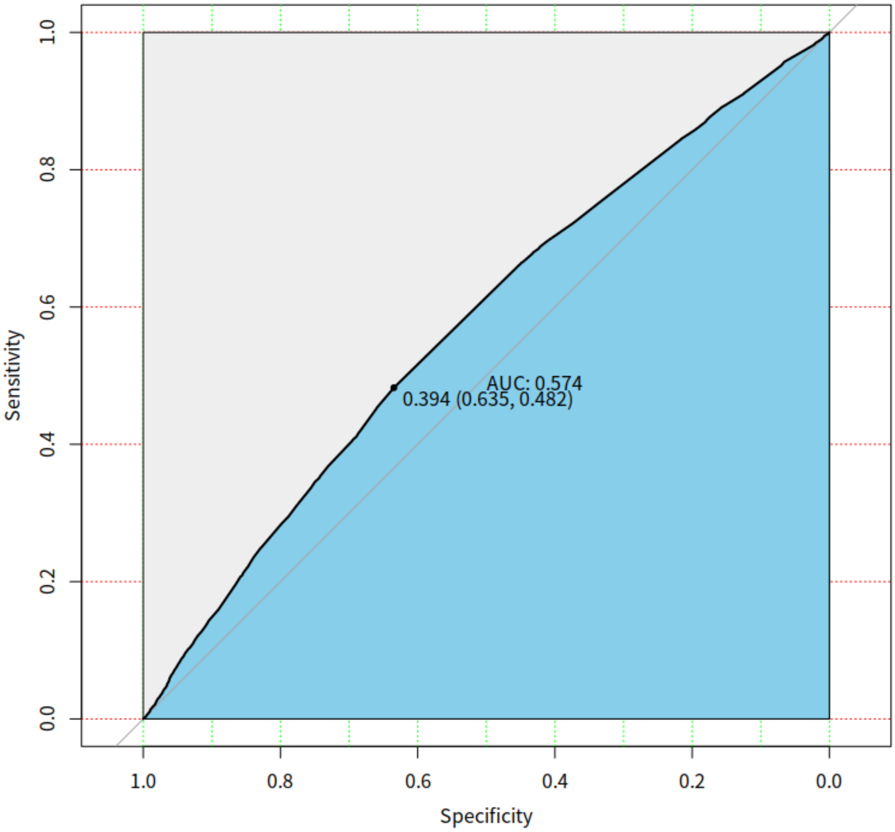

Figure 22:

3. Confusion matrix

```
> test$pre_p<-predict(rpart.mod.pru,test)[,2]
> test$pre=0
> test$pre[which(test$pre_p>0.394)]=1
> print(cft <- table(test$pre, test$y))

       0    1
  0 8364 7565
  1 4814 7052
> tp <- cft[2, 2]
> tn <- cft[1, 1]
> fp <- cft[2, 1]
> fn <- cft[1, 2]
> print(accuracy <- (tp + tn)/(tp + tn + fp + fn))
[1] 0.5546321
> print(sensitivity <- tp/(tp + fn))
[1] 0.4824519
> print(specificity <- tn/(tn + fp))
[1] 0.6346942
```

Figure 23:

Since the AUC=0.574, the classifier obtained from our modeling only slightly outperforms the random guesses. We consider the predictive value of this classifier to be low.
However, according to the confusion matrix, we found that the sensitivity and specificity of this classifier are not high enough.

3.2. K-NN

3.2.1. LOL Dataset

3.2.2. DOTA Dataset

3.3. Naive Bayes

3.3.1. LOL Dataset

Just like the DotA2 data training and evaluation, the training and evaluation phases were performed 100 times. Each time, the F1 score of the model would be calculated. At any point in the iterative process where the current F1 score was better than the previous best-recorded F1 score, the best F1 score and its corresponding data and model were saved. Doing this allowed for us to find the best model out of 100 random samples.

The best model that we were able to generate based on our data performed somewhat well. The following table shows its performance.

LoL Prediction Metrics

F1 Score	0.7512742
Accuracy	0.7530364
Precision	0.7505092
Recall	0.7520408

The following is that same model's confusion matrix.

LoL Prediction Confusion Matrix

	False	True
False	751	245
True	243	737

As can be seen by the above two tables, the best Naive Bayes model we could generate performed reasonably well. With overall scores hovering right near 0.75, this model can more-often-than-not predict the outcome of a game. We suspect that this better performance relative to the DotA2 model is due to the nature of this data set, as it focuses heavily on data collected once 10 minutes have passed in each game. Though this is still usually somewhat early in the game (most League of Legends games tend to average between 25 and 30 minutes at this tier), it would seem that enough has occurred in order to make a reasonable guess as to which team is going to win.

3.3.2. DOTA Dataset

In order to determine a relatively high-quality model, the training and evaluation phases were performed 100 times. Each time, the F1 score of the model would be calculated. At any point in the iterative process where the current F1 score was better than the previous best-recorded F1 score, the best F1 score and its corresponding data and model were saved. Doing this allowed for us to find the best model out of 100 random samples.

The best model that we were able to generate based on our data didn't perform very well. The following table shows its performance.

DotA2 Prediction Metrics

F1 Score	0.6086315
Accuracy	0.5617563
Precision	0.5738121
Recall	0.6479498

The following is that same model's confusion matrix.

DotA2 Prediction Confusion Matrix

	False	True
False	4550	5211
True	3812	7016

As can be seen by the above two tables, even the best Naive Bayes model that we could generate from this data performed poorly. With overall scores of just around 0.6, this model is only barely better than randomly guessing the outcome of a game of DotA2. We suspect that this is primarily due to the nature of the data set that we used, as it relies heavily upon hero selection as opposed to other factors that can only be determined once the game has begun. It would seem that attempting to predict results before matches begin is extremely difficult at best.

4. Comparing

4.1. Horizontal comparison

4.1.1. Decision Tree

From the evaluation results of the model, it can be seen that the decision tree algorithm performs significantly better on the LOL dataset than on the DOTA dataset. Specifically, the decision tree algorithm builds a fully usable classifier on the LOL dataset. However, the classifier built on the DOTA dataset is almost ineffective. Notice that the LOL dataset mainly collects data within the match, while the DOTA dataset collects data mainly outside the match.
If we just consider the differences in data characteristics of the dataset, we can have the following inferences. For MOBAs, the performance of the player while the game is in progress may be more important than the identity of the player, the chosen role. If we look at the data features, the decision tree algorithm seems to perform a little better on the posterior features. But there are similarities between the two classifiers. That is, they both have lower sensitivity than specificity. This means that these classifiers are less prone to errors. This is probably considered an advantage that the decision tree model exhibits.

4.1.2. K-NN

4.1.3. Naive Bayes

We set out to determine if the final match results of two popular Multiplayer Online Battle Arena games, League of Legends and Defense of the Ancients 2, could be predicted based on data from both the start of a game and data from during the game. We used a few different model types in order to perform this prediction, and trained models on two different kinds of data. While our results showed some prediction capabilities, especially from data that was collected during each game, our models did not perform quite as well as we would have hoped.
In training and testing on both pre-game and mid-game data, we aimed to determine if there was a degree of universality between the aspects that the models used on each data type. As can be seen more specifically in the Comparison section, however, there is virtually no universality between the data and their models.
Finally, in using data from multiple games, we aimed to determine if there was any universality between these games that our models could find. [INSERT MORE TEXT HERE]

4.2. Longitudinal comparison

4.2.1. LOL DATASET

4.2.2. DOTA DATASET

5. Conclusion

CRediT authorship contribution statement

Yumin Xu: Programming, Data analysis, Writing - Original draft preparation. **Michael Vigil:** Programming, Data analysis, Writing - Original draft preparation. **Logan Decker:** Programming, Data analysis, Writing - Original draft preparation.

References

[1] Kalyanaraman (2014). To win or not to win? A prediction model to determine the outcome of a DotA2 match. http://cseweb.ucsd.edu/jmcauley/cse255/projects/Kaushik Kalyanaraman.pdf.

[2]Atish Agarwala, Michael Pearce. Learning Dota 2 Team Compositions, 2014.

[3]Thomas E. Batsford. Calculating optimal jungling routes in dota2 using neural networks and genetic algorithms. Project, University of Derby, 2014..

[4]Shuo Chen and Thorsten Joachims. Predicting matchups and preferences in context. KDD, 2016.

[5]Hao Yi Ong, Sunil Deolalikar, and Mark Peng. Player behavior and optimal team composition for online multiplayer games. 2015.

[6]Kuangyan Song, Tianyi Zhang, and Chao Ma. Predicting the winning side of dota2. Course project, Stanford University, 2015.