Matthew Martin, Riley Abrahamson

CSCI 479 Data Mining

Project Report

Introduction

League of Legends is a very popular game of the MOBA (multiplayer online battle arena) genre of video games. It is played with two teams with each team consisting of five champions. Each player in a game controls one champion, so there are ten people in each game. There are currently 146 champions at the time of collecting data, with ten champions selected per match.

Using public data provided by the League of Legends developers (Riot Games), we will be mining a live dataset to analyze the characters available in the game. Our primary goal was to find out the frequent champions in both winning and losing matches. Our secondary goal was to be able to classify matches as a win or loss based off previous matches.

We wanted to find this information because we both play League of Legends and were curious if we could find any discernable patterns within matches in any given patch. With the results we were looking for, we could find out the calculated current best and worst champions and groups of champions to play. This could help with increasing the number of wins a player has and decreasing the number of losses. If we could classify the games, we would be the first people able to do so, which would mean that we could plug in team compositions to the algorithm and see if the composition could be a winning one. This would allow for the hardcore players of the game to optimize their gameplay even further than currently possible.

Methods

We found this information via public API endpoints provided by Riot Games. To gather it, we implemented match gathering and transaction table scripts in Python. To help work with the Riot API, we utilized the open-source community tool *Cassiopeia*, which provided asynchronous calls to the API to help us fetch data.

```
# DATABASE INSERTIONS
while successful_matches < 10000:
    winning_champions = []
    losing_champions = []
    match = cass.get_match(match_id)

    # Check if match_id is valid
    # filter out bot matches
    if match.exists and match.mode.value == "CLASSIC" and match.map.id == 11 and (match.queue.id in (400, 420, 430, 440)):
        # Collect teams
        blue_team = get_champions(match.blue_team)

        red_team = get_champions(match.red_team)

        # For Blue Team wins
        if match.blue_team.win:
            winning_champions = blue_team
            losing_champions = red_team
        # For Red Team wins
        else:
            winning_champions = red_team
            losing_champions = blue_team

        winning_team = {
            'match_id': match_id,
            "winning_champions": winning_champions[0],
            'winning_champions_keys': winning_champions[1]
        }
        losing_team = {
            'match_id': match_id,
            "losing_champions": losing_champions[0],
            'losing_champions_keys': losing_champions[1]
        }

        # Insert winning and losing lists into the databases
        win_champs_col.insert_one(winning_team)
        loss_champs_col.insert_one(losing_team)

        # Increment counter
        successful_matches += 1
        print("Match added!")

        matchfile = open("correct_matches.txt", "a")
        print(str(match_id) + " " +
              str(datetime.datetime.now().time()), file=matchfile)
        matchfile.close()

        if(successful_matches % 200 == 0):
            print("Pausing to avoid rate limit...")
            time.sleep(126)

    match_id += 1
    time.sleep(1)
```

Once data is collected and inserted into the database, it can be withdrawn and formatted for different operations. Our frequency mining called for a transaction table with a header row,

whereas our classifier called for the testing and training files to be match transaction row with an appended class identifier on the end. The format for the transaction tables consist of a row with N columns, N being the number of champions currently in League of Legends. While database entries are pulled out of the database and examined, the program keeps a running list of all champions thus identifier. The order in which champions are identified will correlate with the column number for that champion within the transaction tables.

```
# TRANSACTION TABLE
for match in winning_champs:
    win_transaction = [0] * number_of_champions

    for champ in match.get('winning_champions'):
        if champ not in transaction_table_ids_lookup:
            transaction_table_ids_lookup.append(champ)

        champ_index = transaction_table_ids_lookup.index(champ)
        win_transaction[champ_index] = 1

    winning_champs_transaction_table.append(win_transaction)

for match in losing_champs:
    loss_transaction = [0] * number_of_champions

    for champ in match.get('losing_champions'):
        if champ not in transaction_table_ids_lookup:
            transaction_table_ids_lookup.append(champ)

        champ_index = transaction_table_ids_lookup.index(champ)
        loss_transaction[champ_index] = 1

    losing_champs_transaction_table.append(loss_transaction)
```

```
# Write transactions to files
win_file = open('win_transactions.txt', 'w')
win_file_class = open('win_transactions_classifier.txt', 'w')
print(*[champ.replace(' ', '')
        for champ in transaction_table_ids_lookup], file=win_file)
for match in winning_champs_transaction_table:
    match_classifier = [*match, 1]
    print(*match, file=win_file)
    print(*match_classifier, file=win_file_class)
win_file.close()
win_file_class.close()

loss_file = open('loss_transactions.txt', 'w')
loss_file_class = open('loss_transactions_classifier.txt', 'w')
print(*[champ.replace(' ', '')
        for champ in transaction_table_ids_lookup], file=loss_file)
for match in losing_champs_transaction_table:
    match_classifier = [*match, -1]
    print(*match, file=loss_file)
    print(*match_classifier, file=loss_file_class)
loss_file.close()
loss_file_class.close()
```

We then took the information that Riley gathered and used R code to find the frequent patterns within the matches we took. We used an apriori approach to find the champions that were frequent among the wins and among the losses. The following picture shows the R code that we used to find the frequent subsets. From the code, you can see that we stored the transaction tables into matrices and then took those matrices and called the apriori method from the arules package in R. We then loop over the itemset to create an out-matrix of where each champion is part of a frequent subset.

```
library(arules)

X = read.table("C:/Users/Matthew Martin/Documents/School/Data Mining/Project/win_transactions.txt",header = TRUE)
n = dim(X)[1]
Xp <- as(as.matrix(X), "itemMatrix")
itemSetsX <- apriori(Xp, parameter = list(support= 0.01, target="frequent"))

d = length(itemSetsX) #Number of frequent itemsets
outMatrix = matrix(0,n,d) #trans and frequent itemset.

for(i in 1:d){
  l = as(itemSetsX[i]@items,"list")[[1]]
  print(l)  #itemsets
  Xl = X[,l]
  if (length(l)==1)
    rr=which(Xl==1)
  else
    rr = which(rowSums(Xl) == length(l))

  outMatrix[rr,i] = 1
}

Xneg = read.table("C:/Users/Matthew Martin/Documents/School/Data Mining/Project/loss_transactions.txt",header = TRUE)
nneg = dim(Xneg)[1]
Xn <- as(as.matrix(Xneg), "itemMatrix")
itemSetsXn <- apriori(Xn, parameter = list(support= 0.01, target="frequent"))

d = length(itemSetsXn) #Number of frequent itemsets
outMatrixNeg = matrix(0,nneg,d)
for(i in 1:d){
  l = as(itemSetsXn[i]@items,"list")[[1]]
  print(l)  #itemsets
  Xl = Xneg[,l]
  if (length(l)==1)
    rr=which(Xl==1)
  else
    rr = which(rowSums(Xl) == length(l))

  outMatrixNeg[rr,i] = 1
}

#totalMatrix = rbind(outMatrix, outMatrixNeg, deparse.level = 1)
#print(totalMatrix)
```

After we found the frequent subsets, we moved to classifying the data. We used the python code from assignments 2 and 3 to classify the data using KNN and naïve Bayesian classification respectively. We had to modify our transaction tables to include a class label column so that we could use them for these two methods since they require you to give them whether it was a win or loss. We then took roughly a 90/10 split between training and testing. KNN testing took quite a long time to run due to how many dimensions the data had, so having any larger of a split would have made it less feasible to classify. The following are snippets of code from those two classifiers.

```
for i in range(nn):
    iteration_number = i
    print(i)
    x = Xtest[i,0:d]
    xlabel = Xtest[i,d]
    predictedLabel = 0
    posTest = 1
    for i in range(d):
        posTest = calculate_probability(x[i], meanTrainPos[i], stdTrainPos[i])
        posTest *= posTest
    posCompare = posTest * posClass
    negTest = 1
    for i in range(d):
        negTest = calculate_probability(x[i], meanTrainNeg[i], stdTrainNeg[i])
        negTest *= negTest
    negCompare = negTest * negClass

    if posCompare > negCompare:
        predLabel = 1
    if negCompare > posCompare:
        predLabel = -1

    if predLabel == 1 and xlabel == 1:
        tp += 1
    if predLabel == 1 and xlabel == -1:
        fp += 1
    if predLabel == -1 and xlabel == -1:
        tn += 1
    if predLabel == -1 and xlabel == 1:
        fn += 1


print(nn, tp, fp, tn, fn)
print(accuracy())
print(precision())
print(recall())
```

```
def euclidianDistance(data1, data2, length):
  distance = 0
  for x in range(length):
    distance += np.square(data1[x] - data2[x])
  return np.sqrt(distance)

def getNeighbors(trainingSet, testSet, k):
  distances = []
  length = len(testSet) - 1
  for x in range(len(trainingSet)):
    dist = euclidianDistance(testSet, trainingSet[x], length)
    distances.append(dist)
  neighbors = []
  c = np.argsort(distances)
  for x in range(k):
    neighbors.append(c[x])
  return neighbors

def getLabel(b):
  pos = 0
  neg = 0
  for x in b:
    if x == 1:
      pos += 1
    if x == -1:
      neg += 1
  if pos > neg:
    return 1
  if neg > pos:
    return -1

iteration_number = 0
for i in range(nn):
  f = Xtest[i]
  neighbors = getNeighbors(Xtrain, f, k)
  b = Xtrain[neighbors, d]
  prediction = getLabel(b)
  if prediction == 1 and f[d] == 1:
    tp += 1
  if prediction == 1 and f[d] == -1:
    fp += 1
  if prediction == -1 and f[d] == 1:
    fn += 1
  if prediction == -1 and f[d] == -1:
    tn += 1
  iteration_number = i
  print(i)
```

Results

We found that any combination of champions has very little support in the data. To find a good set of data, we had to reduce the support down to .01 which equated out to roughly 150 matches that the set was found in. The results were that there were no frequent winning combinations with even .01 support. It resulted in only single champion sets. The losses on the other hand, had roughly 40 pairs of champions that were frequent at .01 support. Most of these included the most recent champion Senna, or a popular champion Yasuo. There were 4 champions other than Senna that appeared in 3 or more frequent losing combinations, those were: Miss Fortune, Graves, Yasuo, and Lux.

The classification came back very inconclusive. For KNN classification, we had an accuracy of .4982, a sensitivity of .5571, a specificity of .4334, and a precision of .5177. For the Naïve Bayesian classification, we got an accuracy of .4783. precision of .5, and a recall of .0035

Discussion

The reason that Senna appeared in almost every frequent combination was that she is the most recent champion and thought of to be very strong, so there is a large bias towards her being in more games than any other champion. Yasuo was another champion seen in many of the frequent combinations. He is known for being a popular champion, so again, it makes sense for there to be a bias towards him being in a lot of matches.

Due to the fact that there were 146 available champions, it would make sense that you would have to lower the support needed for frequencies since it is more likely that a champion will not be played than will be played by almost 30 times. This means that one would need hundreds of thousands, if not millions, of matches to find good information. Unfortunately, we were not able to do this due to not have a production level key for the Riot API.

Another problem we ran into because of how the data is set up is that classification becomes very hard to do. If every transaction only has 5 non-zero values and all those values are 1's, then all the data is very clumped together and makes it hard for classification to be done accurately. Classification is also near impossible to do because naturally, all champions have roughly a 50% win-rate. This means that even if you had the amount of data needed to attempt to train, it would only be accurate roughly 50% of the time.

There are currently 2 websites, well known in the League of Legends community, that report frequency data. They are u.gg and op.gg. These websites provide data on how often

champions are picked, banned, win-rate, and many other metrics. These websites are able to provide these services since they have the ability to take every new match and continuously update the numbers. These websites have access to all the resources they could want for being able to classify matches, but they are still unable to do so. They are examples of how classification would be very tough to do for the game due to the nature of win-rates of champions.

Conclusion

The results of our data collection and mining are useful on a hobbyist analytical level but could also prove useful to the developers and balance team behind League of Legends. There are also possibilities for further analysis branching off of this project.

The first takeaway from our results are the balance issues that appear within the frequent champions that appear. If any champions appear frequent, even with a small support (say, 150 games), this could indicate that they may need some balance changes. In our testing, we found the champion Senna to have a high prevalence in our results for both winners and losers. However, this may have to do with the fact that she is a relatively new and popular character.

The next follow-up from our results, is that it may be possible to create a match classifier/predictor of League of Legends matches. While our KNN classifier only held a 50% positive success rate, our dataset was much more limited than what would be available to Riot Games or other data analytical websites like U.GG and OP.GG. Utilizing their larger data stores available, they could hopefully create a more sophisticated classifier.

The last unexpected note from our results is the fact that data mining for frequencies in the manner we did could help with detecting fraudulent players and bot account activity in

League of Legends. While testing our data collected, we came across a some-what high frequency of the team composition Ezreal, Ryze, Galio, Nasus and Alistar. This combination is usual for a team in itself, but this composition only appeared within our losers' frequencies and also had over 50 matches collected of this composition. It is one of the few compositions to reach a frequency of three champions–and infact—all subsets remained frequent up to the enitre 5-player team. Given the usual frequency, we have hypothesized that these could be bot accounts attempt to level up their player profiles in order to be sold later. This would explain the frequency of them appearing together, as well as the simpler (and cheaper) champions that appeared so frequently.

Overall, our final results can lead to conclusions in several directions: development, balance, analysis, and even fraud detection. While we did face limitations within pieces of our data collection and mining, we were still able to generate results and tie them to the current landscape of League of Legends in 2019. While the game is sure to change heading into 2020, this shows a snapshot of how a live service generates patterns and what can be uncovered via mining the Legends.