Learning by Crawling

Program Design and Data Structures (1DL201), 2017/2018

Simon Leijon, Petter Sigfridsson, Jonathan Kurén

Table of Contents

Introduction

Can you predict a winner in a game before it has been played? Surely, it should be possible to a certain degree. By looking at old results you should be able to find some factors that correlate with winning. We have chosen to test this theory on League Of Legends games using a machine learning algorithm. We will be writing the program in the functional programming language Haskell.

Background

League Of Legends

League of Legends is a multiplayer game published by Riot Games. The players of the game are called summoners and you battle other summoners by controlling champions, characters with special abilities. A usual game is played 5 players versus 5 players and the game starts with the players choosing their champion out of a pool containing 139 champions. The players then battle it out with the ultimate goal of destroying the other teams nexus.[1] Additionally, in League of Legends there are different queue types where ranked 5v5 solo is the most competitive one. This is the queue that we look at while fetching information and making predictions explained further down in the report.

Riot API

Riot Games offers an application programming interface, API, where developers get access to League Of Legends statistics. These statistics involve all summoners on all regions and their games played, static data such as all available items, skins and champions and much more. The API is a tool for developers to create platforms & applications for the League of Legends community.

Logistic Regression and stochastic gradient descent

Logistic regression is a model for predicting outcomes where the outcome is binary, such as win/loss or true/false, using one or more predictor variables.[2] By using a method called stochastic gradient descent, which works by predicting an outcome and calculate the error then correcting the coefficients in the algorithm[3], together with logistic regression, you can create a machine learning algorithm.

---

[1] https://en.wikipedia.org/wiki/League_of_Legends
[2] https://en.wikipedia.org/wiki/Logistic_regression
[3] https://en.wikipedia.org/wiki/Stochastic_gradient_descent

The Program

We have created two different programs, a crawler and a machine learning algorithm that will be used to predict League of Legends games. The crawler gathers information from the Riot API and stores it in a JSON file, so it can be used as a data set for the machine learning algorithm. To be able to predict games we need to look at some sort of variable in the games. We have chosen to look at the wins and losses for each player on their specific champion. Why we chose this factor is based on experience playing the game. By only looking at one variable we need to choose one that impacts the game a lot and individual skill is a big part of the game which translates to different win/loss ratios.

Getting Started

In order to get the program running some installations and other preparations are needed. Such as installing all the external libraries mentioned in the *External libraries* subsection and having a valid *League of Legends* developer key from *Riot Games*. Having a stable internet connection is also a requirement to run the program since we are getting data from *Riot Games* application programming interface (API) through the web.

External libraries

All external libraries used in the program are described below. The program requires the user to have these libraries installed on their computer. All of the libraries are installed and downloaded through Haskell's own package manager *Cabal.* Which can easily be used through a command-line interpreter (CLI) of choice. For example windows Command Prompt, MacOS or Linux terminal. The command to install are as follows: `cabal install <Name of library>`. Simply type the command into the CLI to install a library.

Aeson - Data.Aeson v.1.2.4.0

The Aeson library is used to get types and functions for parsing JSON data. Using this library makes it convenient to work with the JSON data our program retrieves from *Riot Games API.* For further information about the library see their documentation[4].

[4] https://hackage.haskell.org/package/aeson-1.2.4.0/docs/Data-Aeson.html

Wreq - Network.Wreq v.0.5.2.0

The Wreq library is used for client-side HTTP requests. The program uses this library to make requests towards the *Riot Games API.* Using this package the response body, headers and status are easily retrieved from a url (the API is accessed through a url). The response body is in our case the JSON data from the url. Response headers contain the request limit towards their API. Response status contains the status code for the request, the status code tells us if an error occurred or if the request was successful. For further information about the library see their documentation[5].

Developer Key - Riot Games

To run our program a developer key from *Riot Games* is required. To make requests towards their API the key is required in the url, see bold text in the example below.

```
https://euw1.api.riotgames.com/lol/summoner/v3/summoners/by-name/sprittiiy?api_key=RGAPI-9322e3b3-ebea-47ff-a298-e4cee9379411
```

In the program the key is used to get access to *Riot Games API* and to construct valid urls to do requests against their API.
The key is acquired through *Riot Games* developer website or as they call it, *Riot Developer Portal*[6]. To get a developer key you must register on their website. After registering and logging in, head to dashboard and then under the section *Development API Key* you can find your personal key.

There are two types of keys: *development keys* and *production keys.* By standard a developer account is granted a *development key* and a *production key* is obtained by registering a permanent project to *Riot Games.* These keys both have rate limits which control the amount of requests a user can do towards their API. See rate limits below.

Development key - by design, very limited and expires after 24 hours.

- 20 requests every 1 second
- 100 requests every 2 minutes

Production key - much larger rate limit than a *development key.*

- *3 000 requests every 10 seconds*
- *180 000 requests every 10 minutes*

[5] https://hackage.haskell.org/package/wreq-0.5.2.0/docs/Network-Wreq.html
[6] https://developer.riotgames.com/

See *Riot Developer Portal* for further documentation about their keys and their API.

Running the program

To run the program some descriptions order. How the modules actually work will be described in further detail in sections The Crawler and Machine Learning.

Initiating the crawler requires you to input an account ID that belongs to a player on Europe West with at least a single ranked solo 5v5 game played. A number of account ID's will be supplied for testing purposes in file DataTypeExamples.hs with the format: accid1, accid2 etc. It is important to note that these account ID's can be changed and there is no guarantee that they can be used indefinitely. After the user has chosen a legitimate account ID the crawler is engaged by calling: *initiateGather "START" accID* - on that account ID. Resuming the gather, similarly, by calling: *initiateGather "whateverStringThatIsntSTART" accID2,* where accID2 is another legitimate account ID. If using the supplied account id's calling, *initiateGather "START" accid1* will suffice. As a side note, it is suboptimal to rerun the crawler on the same account ID multiple times, even though it is possible. This will cause the machine learning algorithm to learn on the same data multiple times and hence make misconstrued adjustments. Additionally, if the crawler is interrupted manually or because of unknown errors the storePlayers.json file needs to be cleared manually before resuming the crawler.

Initiating the machine learning part of the program is accomplished by simply calling function: *startML,* which will read the data from GameStorage.json where the crawler stored all it's fetched data. Before calling *startML* however, a final "]" bracket has to be added to GameStorage.json, for the program to be able to read the file.

Parsing and Custom Data Types

The program is using the external library called Aeson for parsing and handling JSON data, which is also mentioned and described under the *Getting Started* section. Aeson is used to parse the JSON data retrieved through *Riot Games API*. To parse the JSON data using Aeson, custom data types must be created such that they match the structure of the JSON data.

For each of the data types a FromJSON and a ToJSON instance must be created. To decode data, the Instance FromJSON must be written. Similarly the ToJSON instance must be written to be able to encode data. Conveniently there are a language pragma called DeriveGeneric and a instance Generic which lets us write empty FromJSON and ToJSON instances, GHC.Generics

library needs to be imported. The compiler will generate sensible default implementations for those instances.

Creating a data type to match JSON data

The following example is from the documentation of the Aeson library[7].

Given the following JSON data:

```
{ "name": "Joe", "age": 12 }
```

A matching data type can be created as follows:

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics

data Person = Person {
      name :: Text
    , age  :: Int
    } deriving (Generic, Show)

instance FromJSON Person
instance ToJSON Person
```

As mentioned above the language pragma and the Generic instance let us write empty FromJSON and ToJSON instances which are then auto generated. The **Person** data type is created with record syntax. By using the record syntax, Haskell automatically made the following functions for the **Person** data type: **name** and **age**. These functions have the following type:

```
name :: Person -> Text
age :: Person -> Int
```

Extracting data from the **Person** data type, is thanks to the record syntax quite easy. For an example, calling **name** (Person { “Aeson”, 3418 }) would get the output “Aeson”.

[7] https://hackage.haskell.org/package/aeson-1.2.4.0/docs/Data-Aeson.html

Parsing the data

The Aeson library contains a function called *eitherDecode,* the function can be used to decode JSON data into a matching data type. See example of such below.

```
createGameDataObject :: Match -> IO ()
createGameDataObject match = do
  createGameDataObjectTeams match
  c <- (eitherDecode <$> getJSONPlayers) :: IO (Either String [PlayerData]) -- Read file of players
  case c of
    Left err -> do
     print err
     B.writeFile "storePlayers.json" B.empty
    Right players -> insGameDataFile players
  B.writeFile "storePlayers.json" B.empty
```

In this case *eitherDecode* takes *getJSONPlayers* as an argument and is forced to return the output of type **IO (Either String [PlayerData])**. *getJSONPlayers* is of type **IO ByteString**. The **String** represents a error message if the JSON was malformed. The list of **PlayerData** represents the decoded JSON data. After the decoding has been performed the list of **PlayerData** can be accessed, which means that all the JSON data is available to us in Haskell. The **PlayerData** type among other types will be described below.

Data Types

For each JSON data that the program fetches, a data type matching the JSON data is required. The program does only have these kind of data types, but they can be divided into three separate groups. The data types that are used when fetching a match from *Riot Games* API, those used when fetching a players match list. The last group of these are data types that we constructed to be able to create our own .json file. The file are constructed in such a way that there are only information used by the machine learning in the file. All the data types are written with a record syntax. To reiterate, to decode and encode these kind of data types there needs to be FromJSON and ToJSON instances for each of the different data types.

All the data types that are used for matching fetches against *Riot Games* API have the same invariant. The invariant is that all the data in those data types must be valid and come from *Riot Games* API and from the EU West server region. All the data types are located in a own data type module of the name DataFile.hs.

Data types - Match fetch

The program fetches a match and the data are decoded into our data type **Match.** The record type contains **teams, participants** and **participantIdentities** which all are record data types. This means that the **Match** data type contain other record data types. See the figure below.

```
12  data Match =
13    Match { teams :: [Team]
14          , participants :: [Participant]
15          , participantIdentities :: [ParticipantIdentities]
16    } deriving (Show, Generic)
17
18  instance FromJSON Match
19  instance ToJSON Match
20
```

The list of **Team** represents two teams their team id and if they won the game or not. List of **Participant** represent all the participants in the match and their individual participant id, team id, champion id and their rank. The list of **ParticipantIdentities** contain all relevant information about the participants, such as account id, name and summoner id.

The **Team** data type is constructed as previewed in the figure below:

```
24  data Team =
25    Team { teamId :: Int
26         , win :: String
27  } deriving (Show, Generic)
28
29  instance FromJSON Team
30  instance ToJSON Team
```

Participant is constructed as previewed in the figure below:

```
32  data Participant =
33    Participant { participantId :: Int
34                , teamId :: Int
35                , championId :: Int
36                , highestAchievedSeasonTier :: String
37  } deriving (Show, Generic)
38
39  instance FromJSON Participant
40  instance ToJSON Participant
```

ParticipantIdentities data type is constructed as previewed in the figure below:

```
42  data ParticipantIdentities =
43    ParticipantIdentities { participantId :: Int
44                          , player :: Player
45  } deriving (Show, Generic, Eq)
46
47  instance FromJSON ParticipantIdentities
48  instance ToJSON ParticipantIdentities
```

As seen in the figure the data type **ParticipantIdentities** also contains another data type, called **Player**.

The **Player** data type is constructed as previewed in the figure below:

```
50  data Player =
51    Player { currentAccountId :: Int
52           , summonerName :: String
53           , summonerId :: Int
54  } deriving (Show, Generic, Eq)
55
56  instance FromJSON Player
57  instance ToJSON Player
```

Data types - Match list fetch

In the program a players match list is fetched and decoded into our data type **MatchList**. The data type contains a list of **MatchDetails**. This data type is used to retrieve all the games a player has played and makes it possible to e.g. calculate a players win rate with a certain champion. See figure below for the declaration of the data type **MatchList**.

```
59  data MatchList =
60      MatchList { matches :: [MatchDetails]
61                } deriving(Show, Generic, Eq)
62
63  instance FromJSON MatchList
64  instance ToJSON MatchList
```

The data type **MatchDetails** represents one game in a players match list. The data type contain data about the following:

- Which game it is, represented by **gameId.**
- Champion played by the player, represented by **champion.**
- Which queue the game is. e.g. if it is a ranked game or a normal game. This is represented by the **queue.**

See the figure below for the declaration of **MatchDetails.**

```haskell
data MatchDetails =
    MatchDetails { gameId :: Int
                 , champion :: Int
                 , queue :: Int
                 } deriving(Show, Generic, Eq)
instance FromJSON MatchDetails
instance ToJSON MatchDetails
```

Data types - Machine learning

After the program fetches various data and does computations on the data it is convenient to save all the useful information that will be used by the machine learning. We solved this by constructing our own data type **GameData** which we then could encode and save to a .json file with a correct JSON syntax. The **GameData** data type represents its data as two teams, **team1** and **team2.** These two teams are both of the data type **TeamData** which represents its data as five players, **player1, player2, player3, player4** and **player5.** All those five players are of type **PlayerData**. The data type **PlayerData** contain data about the following:

- The players rank, represented by **playerRank.**
- The players losses on a specific champion, represented by **playerChampLoss.**
- The players wins on a specific champion, represented by **playerChampWin.**

See the figures below for the declarations of **GameData**, **TeamData** and **PlayerData.**

Declaration of **GameData**

```haskell
data GameData =
  GameData { team1 :: TeamData
           ,team2 :: TeamData
           } deriving (Show, Generic)

instance FromJSON GameData
instance ToJSON GameData
```

Declaration of **TeamData**

```haskell
data TeamData =
  TeamData { player1 :: PlayerData
           ,player2 :: PlayerData
           ,player3 :: PlayerData
           ,player4 :: PlayerData
           ,player5 :: PlayerData
           } deriving (Show, Generic)

instance FromJSON TeamData
instance ToJSON TeamData
```

Declaration of **PlayerData**

```haskell
data PlayerData =
  PlayerData { playerRank :: String
             , playerChampLoss :: Int
             , playerChampWin :: Int
             } deriving (Show, Generic)

instance FromJSON PlayerData
instance ToJSON PlayerData
```

The Crawler - Farming Data

Since we wanted to have a machine learning algorithm that could predict the outcome of League of Legends games. We quickly realized we would need vast amounts of data to actually make the algorithm learn something. So, we decided to create a crawler. This crawler farms two types of data from the Riot API from each game which is champion win/loss rate and highest achieved rank. We managed to create a type of machine learning described in section Machine Learning. And for this we needed the data.

Module overview

The functionality of the crawler is split into four different modules. Two of these, GameFetch.hs and GatherGames.hs work closely together, performing similar types of tasks but in different ways. Then, there is FetchGate.hs which as the name suggests works as a gate for fetches, not letting the program request any more than the limit to the Riot Game's API. This module is used when functions in GameFetch.hs and GatherGames.hs want to request information from the API. This is necessary so that runtime errors do not occur that would have caused the program to crash. Furthermore, for the program to actually be stable in any long term scenario our

FetchGate.hs module is vital since the requesting privileges would be withdrawn if the program were to cause to many failed requests, disturbing the peace at Riot Games headquarters. Lastly, there is the module DataFile.hs, which quite reasonably contains all the data types used in the various different functions of our program. The structure of these data types is crucial for getting any information to translate from the requested .json files to compilable haskell code in the cooperation with the Data.Aeson package. See section *Parsing and Custom Data types* for more information on this module.

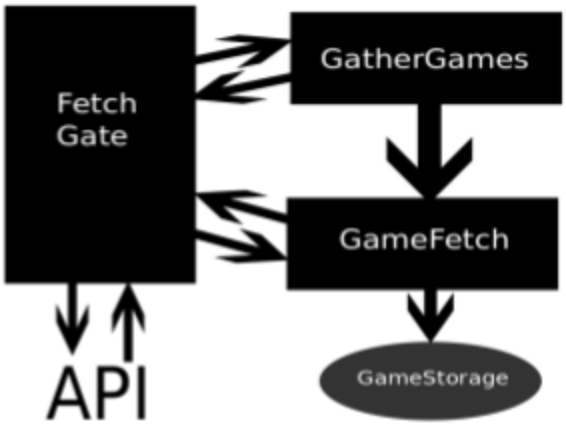


Figure illustrating the flow between modules: FetchGate, GatherGames and GameFetch

Crawler functionality

A simple overview has been presented, now it is time to delve deeper into the functionality of the crawler to get a clearer understanding of what and how things actually work together. Lets begin at the function: GameFetch.hs.

GameFetch

In our machine learning part of the program we want to use data that is combined champion win/loss ratio for each team of a game. Now, to get a combined win/loss ratio for the players on their respective champions, we need to actually calculate the win/loss ratio for each individual player on the champion that they are playing. This is what GameFetch does. One part of the program will create and write a **PlayerData** object to storePlayers.json, containing the needed information for individual players and the other part will combine ten of these together to

construct a **GameData** type containing every player in the game. Illustration with some important functions:

Functions:

```
createGameDataObject :: Match -> IO ()
```

This function is a top level function. In the program it will be called from the GatherGames module but could in principle be called with an arbitrary **Match** type that reflects an actual played League of Legends ranked solo 5v5 game. The first operation of this function is to invoke *createGameDataObjectTeams* which in turn will call *createPlayerDataObject* which writes a **PlayerData** object to a file called storePlayers.json. To not overcomplicate things with different function names we'll go into detail by describing *createPlayerDataObject*.

```
createPlayerDataObject :: Participant -> [ParticipantIdentities] -> IO ()
```

Essentially, this function fetches information from the Riot API on a specific player to then create a **PlayerData** object out of. The information that we need is retrieved by using the players account and champion ID both of which are extractable with the information that the data types in the declaration provide. This information consists of the players entire match list where they played a specific champion. That is exactly the information that the program fetches in this function to then pass on to *calculatePlayerStats*.

```
calculatePlayerStats :: Int -> [Int] -> Participant -> [ParticipantIdentities] -> [Match] -> IO ()
```

This function will receive champId, a match list which will be a list of gameId's ergo **[Int]** where the player has played the champ corresponding to the champId, the same **Participant** and **[ParticipantIdentities]** as the function above and an empty list to accumulate the list of matches. The function will traverse the whole list of gameId's, fetch the match information from each of these from the Riot API and run a set of auxiliary functions on this list of matches to calculate the players champion win/loss ratio and gather other statistics. We call this a set because they all work under the same principle: pattern match the given object/objects and extract the useful information.

<u>The functions that fall under this category are the following:</u>

- $analyzePlayer :: Participant \rightarrow [ParticipantIdentities] \rightarrow (Int, Int) \rightarrow ((Int, Int), String)$
- $analyzePlayerGetAccId :: Participant \rightarrow [ParticipantIdentities] \rightarrow (Int, String)$
- $analyzePlayerGetChampId :: Participant \rightarrow Int$
- $gatherChampGames :: MatchList \rightarrow [Int]$
- $gatherChampsWinLossAux :: Match \rightarrow Int \rightarrow Int$
- $winOrLoss :: Match \rightarrow Int \rightarrow Bool$

- *getAccountId :: Player -> Int*

```
createGameDataObjectTeams :: Match -> IO ()
```

To further describe the upper level functions the function *createGameDataObjectTeams* needs explanation. This function will run *createPlayerDataObject* on every player in the given match (writing the winning team players first) and in doing so will result in storePlayers.json being written the ten players of the match. Additionally this function adds some commas and brackets to the file for correct formatting, which is needed for the program to be able to read the file later on.

Back to createGameDataObject

Now, imagining that, the auxiliary functions have successfully written ten **PlayerData** objects to the storePlayers.json file, *createGameDataObject* uses this file to create a **GameData** object in the file GameStorage.json where we store all our stats. Essentially, it reads the file storePlayers.json, creates a **GameData** object from the reading, writes this to GameStorage.json and finally overwrites the storePlayers.json current information to leave room for the next match to be analyzed.

GatherGames

The GatherGames module makes sure that our GameFetch keeps printing new **GameData** objects to GameStorage by keeping a constant supply of new matches flowing to it. The following functions explain how the module accomplishes this.

Functions:

```
initiateGather :: String -> Int -> IO ()
```

This is the function that will start the whole program. The function takes a string which will determine if the function should write a bracket to GameStorage.json or not and an int which is an arbitrary account ID of a account that has played at least one ranked solo 5v5 game. The account ID given will be sent forward to *gatherMatchSession*.

```
gatherMatchSession :: Int -> IO ()
```

gatherMatchSession will fetch the match list of the given account ID and decode it. Using this match list, it will run two main functions on the information: *createMatches* and *resumeGather*.

```
createMatches :: [String] -> IO ()
```

createMatches takes a list of strings which in this case is a list of url's that are used as API requests to retrieve matches. This list is constructed by first calling *gatherMatchIds* to create a list of match IDs, and then calling *createMatchURL* to construct valid url's corresponding to the request of each match ID. The function will then use this list of url's to create a **Match** object for each url and call our function *createGameDataObject* from module GameFetch. When the function has traversed through the entire list, *gatherMatchSession* will initiate *resumeGather*.

```
resumeGather :: Int -> IO ()
```

resumeGather takes a match ID which will be retrieved by using the first match in the match list created by *createMatches* and picking a new account ID, at random, from the list of players participating in that match and calling *initiateGather* on this new account ID. As such, the crawler will continue to farm data until interrupted.

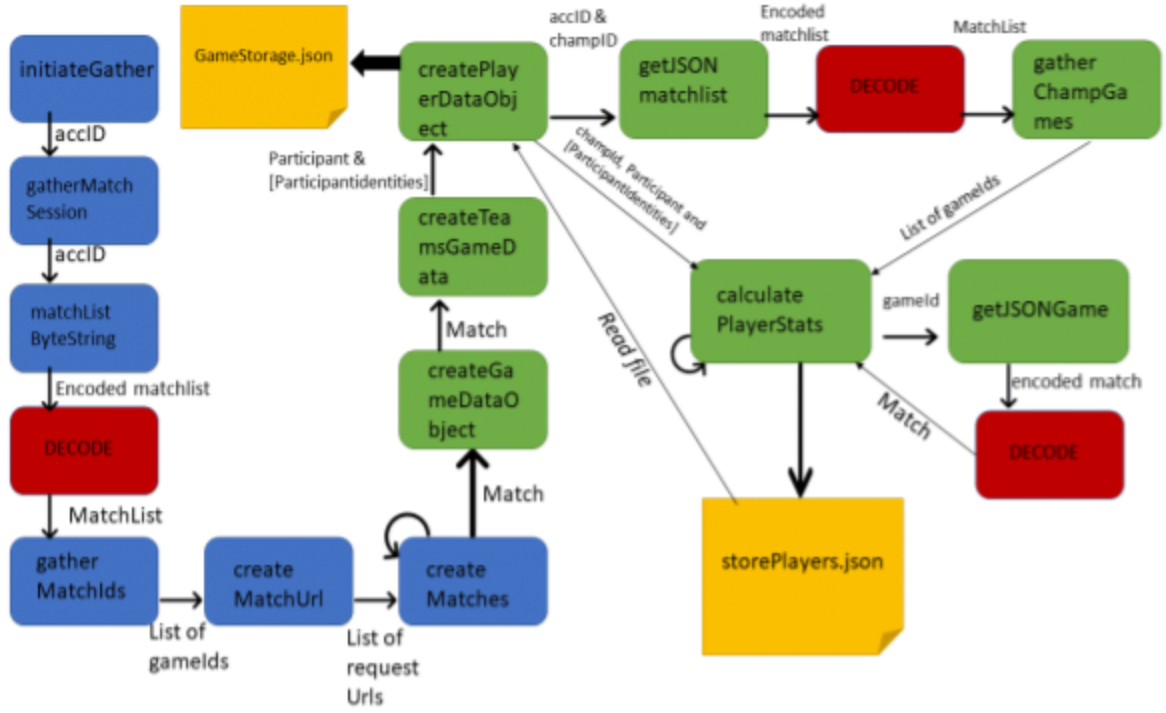


Figure illustrating the flow of the main functions between GameFetch and GatherGames

FetchGate

As explained above the program needs to obey to the request limits of the Riot API and since we do not want our program to crash, nor do we want an angry Riot we have the module FetchGate to resolve this issue.

Function:

```
getContent :: String -> IO BL.ByteString
```

getContent starts by retrieving the status code for the requested fetch (which is the string containing a url), if this code is a 503 or 500 the program will delay for 70 seconds and then resume. In hope that the server error will resolve during this time period. If there was no such error it will check the value of the “X-App-Rate-Limit-Count" header by using the auxiliary function *limitCount*. This function simply returns a bytestring containing the limit count we need to know. If the program is in danger of surpassing that limit we delay the program for 80 seconds, otherwise it continues and returns the body of the fetch (the content we wanted to fetch).

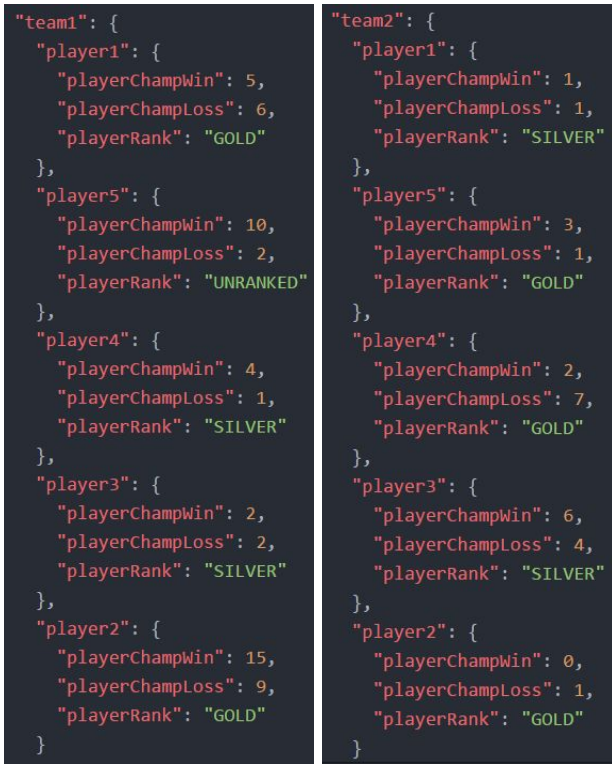
Machine Learning

The second part of this project is machine learning. Since the data we are looking at only has two outcomes, win or loss, we chose to use logistic regression as our model. Logistic regression is often used when the outcome is binary such as, win/loss or alive/dead, since you can represent the outcomes as a 1 or a 0. To train the algorithm a set of data is needed, and all the data used has been gathered by our crawler. There are two modules to the machine learning: Converter, which takes all the data needed and converts it into the right data type and LogisticRegression, which uses this data to train the algorithm.

Converter

The converter starts by fetching all the data from a JSON file and divides the data into two different sets, the training set and the test set. Where the union of the sets equals the empty set. The logistic regression uses [(Double,Double)] and the data set is [GameData] which means the data has to be converted. The two data points we are returning to the logistic regression is the ratio in the two teams combined wins and losses and the winning team, represented as a 1 or a 0. To get the first data point we need to look at the two **TeamData** in **GameData**. Each **TeamData** consists of 5 **PlayerData** and to get a team’s win loss ratio we simply add all *playerChampWin*

together and all *playerChampLoss* together and divide the two. Then, when both teams have their ratio we divide their respective ratios with each other to get the final ratio. Example:



```
"team1": {
  "player1": {
    "playerChampWin": 5,
    "playerChampLoss": 6,
    "playerRank": "GOLD"
  },
  "player5": {
    "playerChampWin": 10,
    "playerChampLoss": 2,
    "playerRank": "UNRANKED"
  },
  "player4": {
    "playerChampWin": 4,
    "playerChampLoss": 1,
    "playerRank": "SILVER"
  },
  "player3": {
    "playerChampWin": 2,
    "playerChampLoss": 2,
    "playerRank": "SILVER"
  },
  "player2": {
    "playerChampWin": 15,
    "playerChampLoss": 9,
    "playerRank": "GOLD"
  }
```

```
"team2": {
  "player1": {
    "playerChampWin": 1,
    "playerChampLoss": 1,
    "playerRank": "SILVER"
  },
  "player5": {
    "playerChampWin": 3,
    "playerChampLoss": 1,
    "playerRank": "GOLD"
  },
  "player4": {
    "playerChampWin": 2,
    "playerChampLoss": 7,
    "playerRank": "GOLD"
  },
  "player3": {
    "playerChampWin": 6,
    "playerChampLoss": 4,
    "playerRank": "SILVER"
  },
  "player2": {
    "playerChampWin": 0,
    "playerChampLoss": 1,
    "playerRank": "GOLD"
  }
```

Team1:(5+10+4+4+15)/(6+2+1+2+9)=7/5
Team2: (1+3+2+6+0)/(1+1+7+4+1)= 6/7
Total: (7/5)/(6/7)=5/6

GameData represents the winner of the game as team1 and the information for the regression will be converted to (5/6, 1.0) or (6/5, 0.0), more on the two different cases later in the documentation.

How does it convert?

The program starts by calling the function *startML*, which fetches the data set from a JSON file. Then it divides the set into a training set and test set by using *trainingTestDivider*. The training set is then converted into weights using *mainPredictor* (see Logistic Regression). The total accuracy of the program is then calculated using the weights and the test set in *calculateAccuracy* (see Logistic Regression). To convert the data into **[(Double,Double)]** we use *getStatistics*, which takes each **GameData** in the set and converts it into **(Double,Double)** and works through the **[GameData]** recursively. The auxiliary function *statisticsAux* converts each **GameData** into **[PlayerData]** in *toPlayer* and thereafter converting each **PlayerData** into **(Double,Double)** using *champWinLoss* where the tuple consists of total amount of wins and total amount of losses. "***(Double,Double) = (wins, losses)***". When the whole set has been converted

into **[(Double,Double)]** we have to attach the winner. This is done in *comparisons*, and to get the logistic regression to work we need a training set where there are both 1s and 0s as results. Since we know that team1 is the winner we need to alternate if team1 is first or last in the comparison. Therefor the same game can be represented two different ways. This way we get a data set where there are as many 1s as 0s. This function returns the final list that is input for the regression, with the type **[(Double,Double)]** where the first value is the ratio between the two teams win loss ratio and the last value is the winner represented as a 1 or 0.

Functions

```
startML :: IO ()
```

Takes all the games from a JSON file and calculates the accuracy using logistic regression

```
trainingTestDivider :: Double -> Int
```

Calculates 80% of the input value, used for splitting up a data set

```
comparison :: [(Double,Double)] -> [(Double,Double)]
```

Calculates the ratio between the two team win loss ratio and alternates between games if the winning team is team 1 or team 2, so the logistic regression can see two different outcomes. If we don't alternate the result will always be 0, since team 1 is the winner, and the regression will not work.

```
getStatistics :: [GameData] -> [(Double, Double)]
```

Creates a list of tuples where each tuple has the statistics which will be used in the logistic regression

```
statisticsAux :: GameData -> (Double, Double)
```

Creates a list of tuples where each tuple has the statistics which will be used in the logistic regression

```
champWinLoss :: [PlayerData] -> (Double,Double) -> Double
```

Calculates the ratio in a team's total wins and losses

```
toPlayers :: GameData -> ([PlayerData], [PlayerData])
```

Takes all the players out of the game

Logistic Regression

To create our machine learning algorithm, we used logistic regression and SGD(stochastic gradient descent). The regression is used to make prediction based on input variables, in our case the ratio between two teams win/loss ratio. The SGD is used to calculate new weights in the algorithm based on the prediction, input, actual result and learning rate. The learning rate is something you have to experiment with and we found 0.4 to be a good number.
There are two parts to this module. One that calculates optimal weights based upon a training set and one that calculates how well the program predicts game outcomes using these weights.
The training part of the program starts in *mainPredictor* where you input a list of tuples containing the parameter and the result of the game. The function is recursive and uses the head of the list in each recursive call. The parameter is used in the *helpFunction* together with the current weights (All weights start at 0) and the output is then used as an argument in the sigmoid function *logPredictor*. The output of the sigmoid function will be the algorithms guess on who won the game, a number between 0 and 1. The new weights are then calculated in *updater* where you input the algorithms guess, the result and the old weights. The new weights are then calculated and used for the next recursive call in *mainPredictor*. When the list is empty the final weights become output and those will be used as the trained algorithms weights.

The test part of the program starts in *mainFunction* and takes a similar list of tuples plus the weights calculated by the training. The function *totalPredicts* then recursively goes through the list and uses the parameter and the weights in the *helpFunction* which is then inputted as the argument for the sigmoid function *logPredictor*. The output of the sigmoid function is the trained algorithms guess and when the algorithm has made a guess on all values in the list, all values are then rounded up to 1 or down to 0 using *converter*, depending on if they are smaller or bigger than 0.5. The function *correctPredicts* is then used to calculate how many correct predictions the algorithm made and *accuracy* then calculates the “correct predicts” percentage.

Functions

```
logPredictor :: Double -> Double
```

The sigmoid function which will calculate the prediction and output a result between 0 and 1

```
helpFunction :: Double -> Double -> Double -> Double
```

Calculates the value which will be the input of the logistic function using weights and chosen parameters

```
mainPredictor :: [(Double,Double)] -> (Double,Double)
```

The function which calculates the final weights based on the training set

```
updater :: Double -> Double -> Double -> Double -> Double
```

Calculates the new weights based upon old weights, the algorithms prediction and the actual result. The updater also has a learning rate which the tester has to configure based upon testing. We discovered that a learning rate at 0.4 gave the best results.

```
totalPredicts :: [(Double,Double)] -> Double -> Double -> [(Double,Double)]
```

Makes a list of all the predictions in a test set by inputting a test set and the weights from the training

```
converter :: [(Double,Double)] -> [(Double,Double)]
```

Converts numbers in a list to either 1 or 0 depending on if they are > or < than 0.5

```
correctPredicts :: [(Double,Double)] -> Double
```

Determines how many correct predicts you have

```
accuracy :: [(Double,Double)] -> Double -> Double
```

Gives a percentage of how many of the predicts were correct

```
mainFunction :: [(Double,Double)] -> Double -> Double-> Double
```

Calculates the accuracy on a test set with given weights

Results: Our dataset is far to small to get an accurate result. We managed to farm 190 games and by running the algorithm on these games we got a predict accuracy at 89%. This however does not mean our algorithm has a 89% chance to predict games, since the dataset is far to small and outlier results have to much of an impact.

Discussion

There are several areas in which our program could be improved to make it more useful, effective and stable. The following are weaknesses we which to highlight to further explain what it is that could make the program useful and what it is that hinders the possibility of that.

The Game Prediction

The motive for creating the program described in this report was to be able to predict a live League of Legends game based on the players in the game and their individual statistics. While we managed to create a crawler to farm data that we could use to train an algorithm to make predictions, we fell short on making it applicable on live games. This is partly due to restrictions of available information through requests during a live game and partly due to us not having the time to work on this implementation.

Small Data Set

As mentioned previously, the data set that we use to train our algorithm remains fairly small despite the crawlers best efforts. The limits set by Riot themselves hinders us from gathering large amounts of data without being delayed periodically. Without large amounts of data it is uncertain how precise our algorithm that makes the predictions actually is.

Variables

Even though we tried to direct our focus at the one variable that affects the outcome of a League of Legends game the most and therefore would be the source of most other variables there are undoubtedly other variables that we do not take into consideration in our algorithm. This in turn makes it impossible for the algorithm to achieve maximum efficiency regarding predicting outcome of games.

Formatting

When we write information to .json files the data needs to be written in a specific manner so that we can actually read the file when we need to. The program will add the appropriate characters to the file during runtime but when the user wants to use the file containing the GameData objects they need to manually add the last bracket to the file to be able to read it as a .json-file. Furthermore, if the program runs into an unknown error or if the user manually disrupts the program during runtime the storePlayers.json file will not be cleared and will therefore hinder the program from running properly, once reinitiated, if the file is not manually cleared.

Undesirable requests

Our functions that fetch match lists and specific games run the risk of needing to fetch the same data multiple times. This is not optimal, considering the limitations put on requests and our program computing unnecessary calculations.

Further Development

Cycling keys & regions

The Riot API allows users of the API to make the same amount of requests on every server (Europe West, North America, Oceania etc..). This means that further development could imply the use of this possibility to gather up to ten times more data at the same rate that we are farming right now. In addition to this we are only using one developer API-key to make requests while we could be using three simultaneously or switching between them when we hit the request limits. Implementing these improvements would at least solve our issues of having small data sets to train our logistic regression algorithm with.

Interface

An enhancement of our program would be to develop some sort of interface that can force the program to execute certain commands. This could be used to be able to stop the crawling without having to manually change the files that store information.

Combining Functions

We have several functions that compute the same type of value or fetch the same type of data from Riots API. An example of this is the function matchListByte and getJSONmatchlist. These could be combined with some effort and would result in more readable code that requires less functions.

Caching retrieved information

Lastly, we could solve the issue of having to fetch the same data multiple times by using some sort of caching mechanism where we cache the information locally for a given amount of time. Since the crawler is intended to run boundlessly we would not be able to cache all the information. This, however, would not be needed since the data that is likely to be needed multiple times will lie in close proximity based on the nature of our way of picking matches to analyze.