Danmarks Tekniske Universitet

02807 Computational Tools for Big Data

Big Data Study on RIOT GAMES

Teacher and teaching assistant:

David Kofoed Wind,

Finn Årup Nielsen

(Rasmus) Malthe Jørgensen

Ulf Aslak Jensen

Preface

League of Legends(LOL) is an online battle video game developed by RIOT Games, popular in Europe, North America and East Asia. Since its publication, LOL has drawn public attention for its novelty and popularity. Luckily, RIOT Games provide abundant APIs for researchers to study unknown laws in the game. In this project, hence, we will mine the data through APIs and analyze the performance of different summoners or champions, during which process several skills learned in this course like Apache Spark, MongoDB, Map Reduce and DBSCAN clustering will be utilized.

Introduction

For a clear structure, the project report is divided into several parts. **Project description** gives a detailed report of the background and what we are going to do with data, telling the specific methods used. **Problem formulation** talks about two main problems we are faced with and key skills to solve them. **Design Specifications** illustrates the structure of the dataset, giving our main idea of the project. **Implementation** is about code but not only the code. This part also combines some notes and description. Finally in **Conclusion** we would review the whole project. Some original codes and plots are attached in **Appendix**.

Project Description

RIOT gaming has launched a game called League Of Legends (LOL) since 2010. It is a 2D online competitive game that summoner use the characters (champions) to take down their opponents. As the popularity of the game increases, the generated data sets become huge by the increasing summoner game behaviors.

The RIOT gaming offers an organized API for developers to query the database to registered developers. The API documentation shows a full reference of the data sets. By requesting the specific API, developers can fetch related information as they wish to analyze.

In this project, we are going to mine the data of game statistics information based on given summoner names in the top two leagues. Our group studied how to use the RIOT API to get the related JSON formatted data, for example, the champions' (the characters in the game) win-rate in recent games played by summoners (the game players). The fetched data will be organized by IDs (unique in the game) of each summoner per JSON file, containing their game statistics. We will fetch approximately hundreds of thousands of summoners' game information.

The main goal of our project is to retrieve and analyze RIOT game statistical large data sets. The size of the datasets is approximately 8GB.

Our study contains:

- the recursive data retrieved by using RIOT API,
- Apache Spark job for parallel operation,
- MongoDB queries and aggregates,
- DBSCAN clustering analysis,
- and visualizing the statistical result based on the data.

The dataset is fetched by a home-made library. After downloading the data to local files, we make some initial plots and network analysis to test on part of our data. Then we use spark job to create Resilient Distributed dataset(RDD), and make parallel operation on a cluster to fetch new datasets. Afterwards we save the new datasets which specified into MongoDB for later analysis. Then we will plot the data and show the values to give direct visualization. Finally, we will do DBSCAN clustering analysis.

Problem Formulation

The first problem for our project is retrieving gigabytes level of data set from RIOT database. The RIOT API has a call limit, with 10 seconds no more than 10 requests, we used time.sleep() function to deal with this problem. Besides, the game API does not have a full list of player IDs, which means we have to retrieve our own player ID list.

Unlike some other projects, downloading game data using API is quite a challenge already from the beginning stage. The relatively large datasets among RIOT database is the game table. It mainly contains the game stats of 10 summoners' activity. There are many nested keys with related data which stored as JSON dictionary format. We will collect the summonerID and query the game API by the collection of summonerID. The server will respond with recent 10 games' stats of the summonerID. In each game stats, there are 9 fellow players. We will add the 9 fellow players to the collection of summonerID. In this way we will expand the collection of summonerID. The amount of the game datasets would be increased consequently. These datasets will then be stored in local drive.

The second problem we encountered is to process hundreds of thousands of game-datasets.

We will use spark job RDD API and MongoDB to transform the multiple game datasets into new datasets of championIDs and its stats over certain game. The stats will include for e.g. win/lose, damage dealt by the champion played by certain summoner. Then datasets will be reduced by each championID and the aggregation of its various game stats.

The third problem we are faced with is to analyze the statistical data and plot them.

As for this problem, we will use the plot library to plot the aggregation of each game statistics over all champions. The plot will show a specific game stat on the y-axis and all champions on the x-axis.

Design Overview

The chapter shows our design in a top level. It contains the overview of the design.

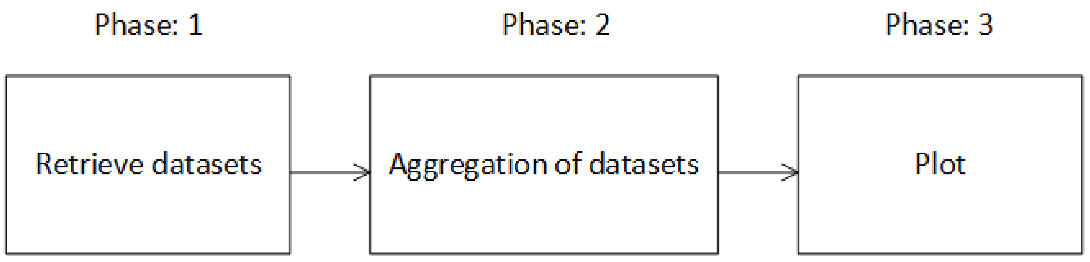


Figure above shows the top level block diagram of the design. It refers the problems listed in the problem formulation.

In the phase 1, we design a class for all RIOT API request.. These API are summoner, game and champion. We will come back to explain more specifically what we can fetch by doing so.

In the phase 2, we collect game statistics for each champion and aggregate each stat. Then we collect these new datasets and save them into MongoDB.

In the phase 3, we use the aggregated data to do some calculations for e.g. to calculate the win rate of each champions, the frequency of each champion that all summoners has used and the average of the performance of the champions . These calculation will then be plotted as each datasets verse all champions.

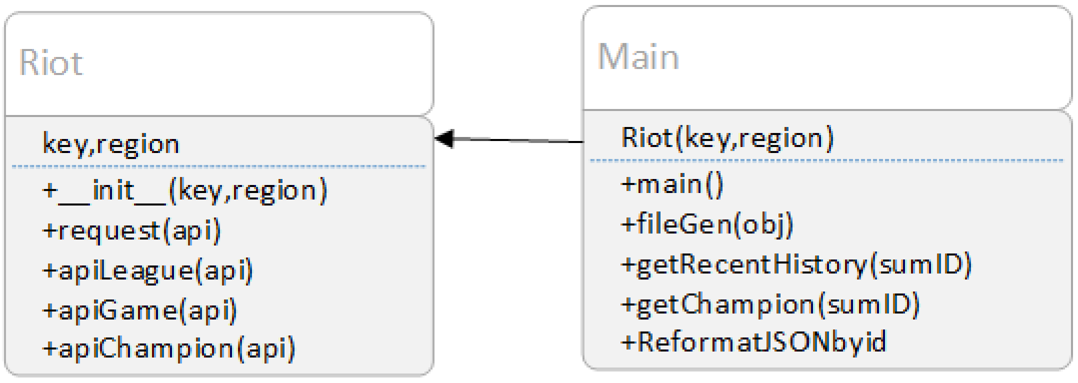
Design Specifications

The chapter will show the detailed design specifications for each phase which has been showed in the previous chapter.

Phase 1:

In this phase we will make a class for RIOT API requests which are necessary for us . As we talked about in the design overview, we will query summoner, game and champion APIs. This section explains the structure of the class and each API in a top-down level

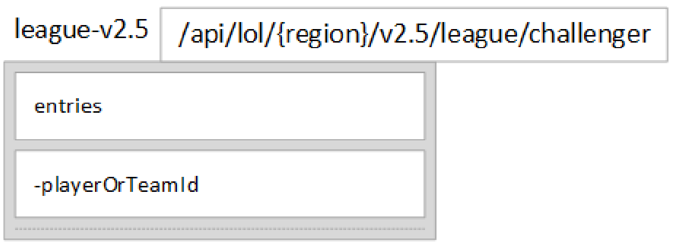
Class diagram



The Riot class has a constructor with key and region. It offers 3 operations which are 'apiLeague', 'apiGame' and 'apiChampion'. In the main class we will initialize the Riot class and use the functions of Riot class to download data sets.

1. League API

This API will be used in a python function for retrieving the offered summonerIDs that RIOT offers in the two top leagues. We will collect these IDs to request the game API that will returns larger datasets of game stats.



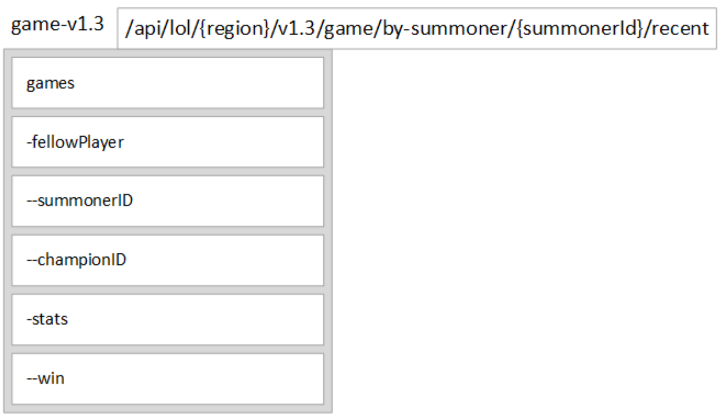
The figure above shows the league API version 2.5. In the figure, 'league-v2.5', /api/lol/{region}/v2.5/league/challenger', 'entries' and '-playerOrTeamId' shows the name of the RIOT API, general content of players in challenger league request, top level key 'entries' and lower level key 'playerOrTeamId'. The entry of key 'playerOrTeamId' contains the summonerID.

https://{region}.api.pvp.net/api/lol/{region}//euw/v2.5/league/challenger?api_key={developer API key}

The url above is a model of challenger league request. All the content inside curly bracket are the arguments. In the url, 'region' and 'API key' indicates game server region and registered API key.

2. Game API

The request will be used for two purpose. The first one is used for retrieve the offered fellow players which played with a certain summoner. We will collect these IDs to expand the original summonerID collection. The second is to retrieve the game stats of championID and all stats. The game datasets will be expanded in gigabytes level to as more summonerID as we can collect.



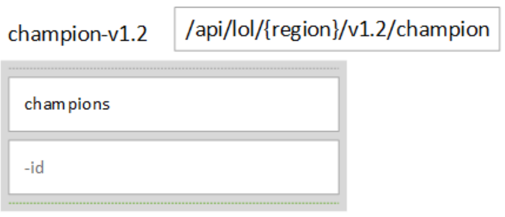
The figure above shows the game API version 1.3. In the figure, 'game-v1.3', '/api/lol/{region}/v1.3/game/by-summoner/{summonerId}/recent', 'games' ,'fellowPlayer', '--summonerID', '-stats', '--win' and '--K/D/A' shows the name of the API, general content of the game data request, the top level key 'games', the lower level key 'fellowPlayer', the 2nd lower level key '--summonerID', the lower level key '-stats', the 2nd lower level key 'win' and the 2nd lower level key 'K/D/A'

```
https://{region}.api.pvp.net/api/lol/{region}/v1.3/game/by-summoner/{summonerID}/recent?api_key={API key}
```

The url above is a model of challenger game stats request. All the content inside curly bracket are the arguments. In the url, 'region', 'summonerID' and 'API key' indicates game server region, specific summonerID and registered API key.

3. Champion API

The request will be used to collect the championIDs. The championIDs are unique. We will be used the championIDs as our analyzed object.



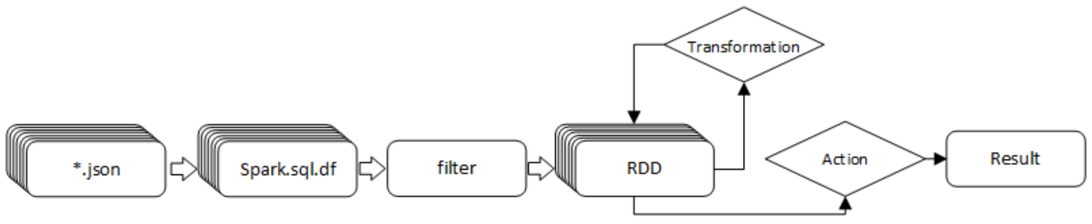
The figure above shows the champion API version 1.2. In the figure, 'champion-v1.2', '/api/lol/{region}/v1.2/champion', 'champions' and '-id' shows the name of the API, general content of the champion data request, the top-level key 'champions' and the lower level key 'Id'.

```
https://{region}.api.pvp.net/api/lol/{region}/v1.2/champion?api_key={API key}
```

The url above is a model of championID request. All the content inside curly bracket are the arguments. In the url, 'region' and 'API key' indicates game server region and registered API key.

Phase 2:

In this phase, we will use Apache Spark to process all JSON datasets which contains game stats. The spark map-reduce method is used to fetch related data. For e.g. champion appearance, win or lose statistics for individual champion and so on.



The figure above shows the spark process applied to the project. The process begins with converting all json files to spark sql data frame. The next is to filter out each specific data frame and then convert to RDD for later use. Then we will have a ready RDD for map each key with value in the transformation stage. After that we will reduce the datasets to each unique key and aggregated value in the Action stage. Finally we will save the result to file. The detailed implementation can be found in the Spark Map-reduce chapter.

We will store the data into MongoDB, and do some queries using things we've learned. After some simple queries, we will group our results using aggregation methods, to see some statistical results. Finally, we will store our results, and visualize results using multiple kinds of plots.

Phase 3:

As for the result visualization, we will try to format our results in a comprehensible and vivid way. For example, using tables or graph database to present the query results.

Besides, we will consider many game players and the champions they choose in this project. So we will use multiple kinds of plots to present different kinds of results, like histograms, scatter plots, graph plots, etc. You will find more about our plots in the later part!

Implementation

1. Riot class

As we describe above The riot class offers three operations such as 'apiLeagure', 'apiGame' and 'apiChampion'. The Riot class imports a customized constant file which contains the API constants. The full code is included in the appendix. This section shows some examples of how the class is used.

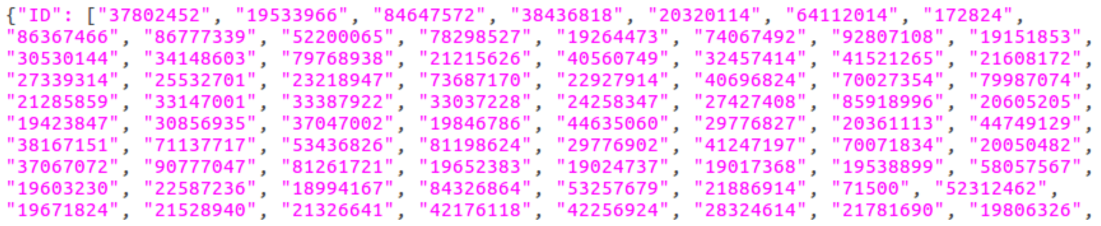
```
class Riot:

##class constructor
def __init__(self, key, region):
    self.key = key
    self.region = region

##basic request
def request(self, api):
    url = const.API['base'].format(R=self.region, A=api, K=self.key)
    r = urlopen(url)
    jobj = r.read()
    pobj = json.loads(jobj)
    return pobj

### get challenger league summoner info
def apiChallenger(self, api):
    self.api =api.format(V=const.API_VERSION['league'],T=const.MATCH_TYPE['1'])
    return self.request(self.api)
```

The code above is part of the Riot class which shows the class constructor takes 'key' and 'region'. The 'request' function will form a complete URL request by specified the API. It will convert the JSON object to python object.

```
### main function
def main():

### initialize the Riot class with give key and region
riot = RiotQ(const.KEY['wen'], const.REGION['europe_west'])
### call the apiChallenger function and save the datasets to challenger
challenger = riot.apiChallenger(const.API['challenger_league'])

### generate a file to save summonerID dictionary
idFile(getID(challenger), 'w')
print getID(challenger)

if __name__=="__main__":
```

```
main()
```

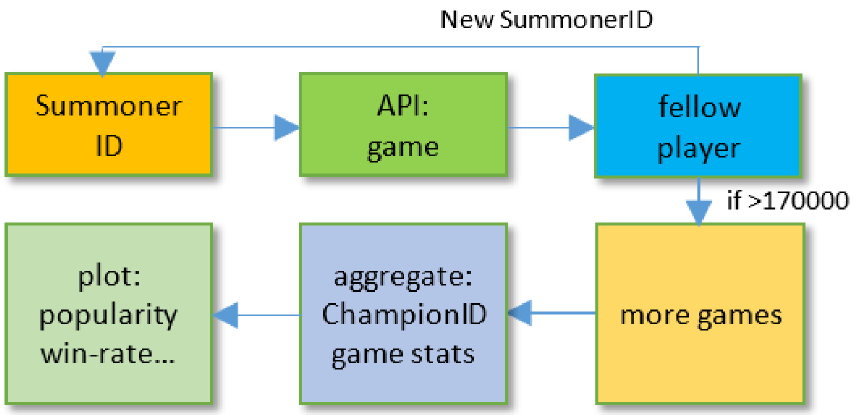
The code above would output a file of summoner ID. This is done by using the ‘apiLeague’ operation from Riot class.



```
{"ID": ["37802452", "19533966", "84647572", "38436818", "20320114", "64112014", "172824",
"86367466", "86777339", "52200065", "78298527", "19264473", "74067492", "92807108", "19151853",
"30530144", "34148603", "79768938", "21215626", "40560749", "32457414", "41521265", "21608172",
"27339314", "25532701", "23218947", "73687170", "22927914", "40696824", "70027354", "79987074",
"21285859", "33147001", "33387922", "33037228", "24258347", "27427408", "85918996", "20605205",
"19423847", "30856935", "37047002", "19846786", "44635060", "29776827", "20361113", "44749129",
"38167151", "71137717", "53436826", "81198624", "29776902", "41247197", "70071834", "20050482",
"37067072", "90777047", "81261721", "19652383", "19024737", "19017368", "19538899", "58057567",
"19603230", "22587236", "18994167", "84326864", "53257679", "21886914", "71500", "52312462",
"19671824", "21528940", "21326641", "42176118", "42256924", "28324614", "21781690", "19806326",
```

The figure above shows the ‘summonerID’ in JSON format

2. Download Data



We use RIOT game API to download all the data we need. The API full reference is: https://developer.riotgames.com/api/methods.

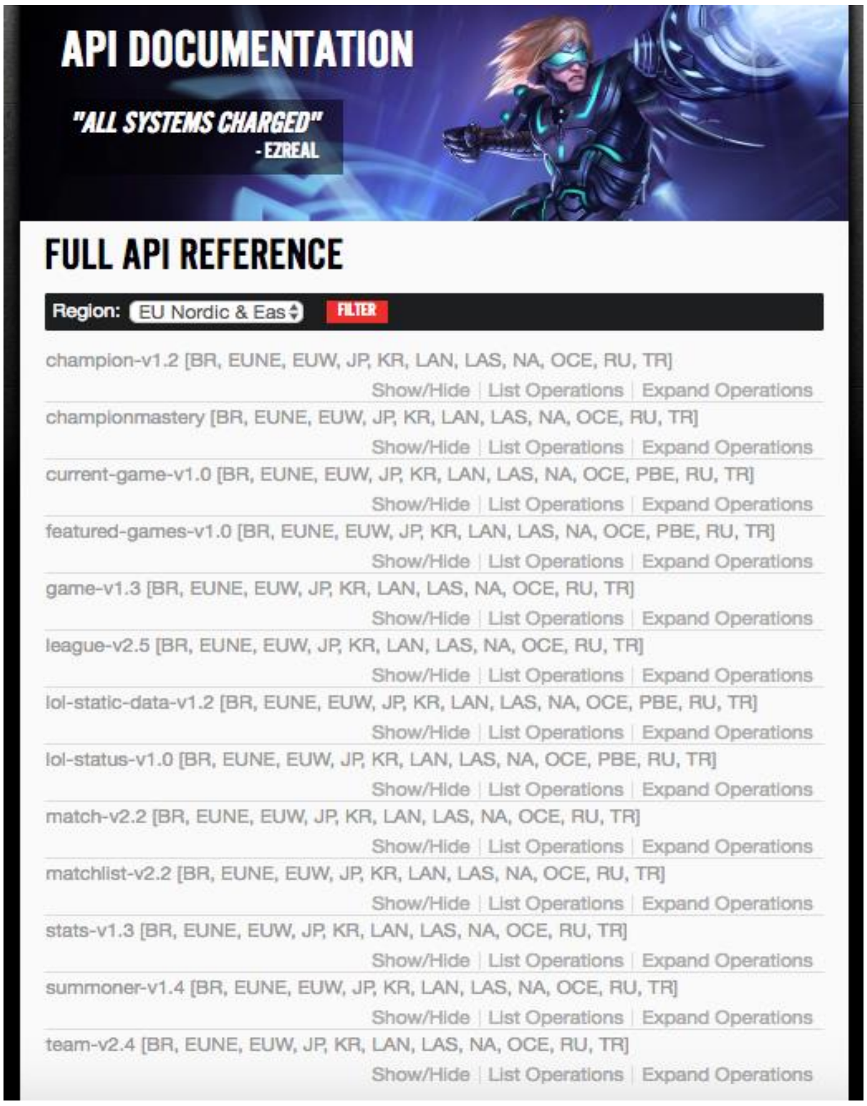
Because RIOT doesn’t have a list of all the players, we used a Summoner Name randomly chosen from the RIOT top league, retrieving his user information in order to find more summoner ids.

As the graph above shows, we first get one initial Summoner ID. Then, by querying the game-v1.3 api, we get the recent 10 games of this summoner. For each game, we can get the Summoner ID of 9 fellow players. We extend the IDs newly found to the Summoner ID list. Then by iterating these steps, we can get a large amount of Summoner IDs. And we store them in a file.

First, we implement all the libraries we need. Then, we defined several functions, in order to get the different APIs and store the json data to our local file. Below is the sample code demonstrating how we get JSON reply and use API to get a summoner's recent 10 games, and store the json results into a local file.

```
# Get JSON reply using the URL
def getJSONReply(URL):
   response = urllib2.urlopen(URL)
   html = response.read()
   data = json.loads(html)
   return data

# Using the game-v1.3 API to get each player's recent history
def getRecentHistory(SummonerID):
      rURL= "https://" +Region.lower()+ ".api.pvp.net/api/lol/" + \
      Region.lower()+ "/v1.3/game/by-summoner/" + `SummonerID`+ "/recent?api_key=" + Key;
      r_data=getJSONReply(rURL);
      r_data['_id']=r_data['summonerId'];
      r_data.pop('summonerId');
      # Write the data to a local json file, naming it after this Summoner's ID
      with io.open('RecentHistory/%s.json' % str(SummonerID), 'w', encoding='utf-8') as f:
         f.write(unicode(json.dumps(r_data, ensure_ascii=False)))
      return r_data;
```

Here is a screenshot of the RIOT API full documentation.



FULL API REFERENCE

Region: EU Nordic & Eas FILTER

champion-v1.2 [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, RU, TR]
Show/Hide | List Operations | Expand Operations

championmastery [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, RU, TR]
Show/Hide | List Operations | Expand Operations

current-game-v1.0 [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, PBE, RU, TR]
Show/Hide | List Operations | Expand Operations

featured-games-v1.0 [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, PBE, RU, TR]
Show/Hide | List Operations | Expand Operations

game-v1.3 [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, RU, TR]
Show/Hide | List Operations | Expand Operations

league-v2.5 [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, RU, TR]
Show/Hide | List Operations | Expand Operations

lol-static-data-v1.2 [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, PBE, RU, TR]
Show/Hide | List Operations | Expand Operations

lol-status-v1.0 [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, PBE, RU, TR]
Show/Hide | List Operations | Expand Operations

match-v2.2 [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, RU, TR]
Show/Hide | List Operations | Expand Operations

matchlist-v2.2 [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, RU, TR]
Show/Hide | List Operations | Expand Operations

stats-v1.3 [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, RU, TR]
Show/Hide | List Operations | Expand Operations

summoner-v1.4 [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, RU, TR]
Show/Hide | List Operations | Expand Operations

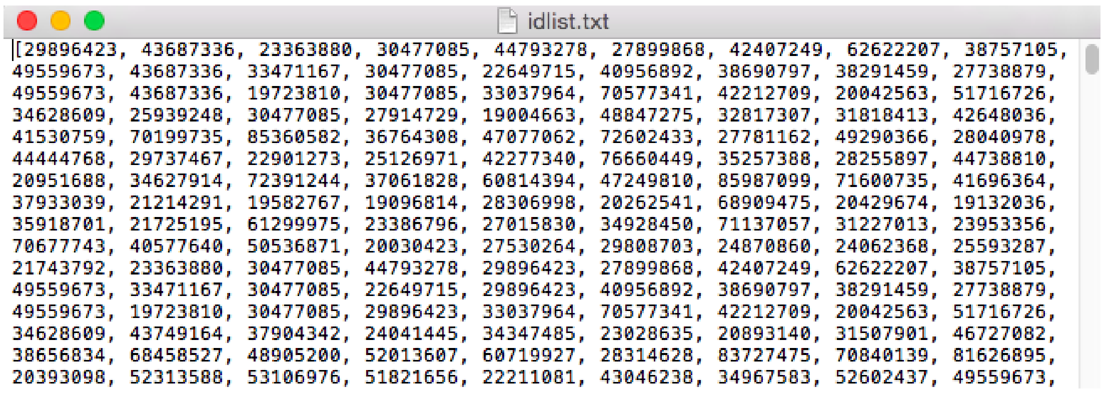
team-v2.4 [BR, EUNE, EUW, JP, KR, LAN, LAS, NA, OCE, RU, TR]
Show/Hide | List Operations | Expand Operations

Using the functions we've defined, we can get API reply in a structured way.

```
# Pre-input using SummonerName provided by RIOT top league
SummonerName="Miladena"
Region="EUW"
Key="RGAPI-20253dda-c325-4a7e-947a-d283af4f8641"

# Retrieving the SummonerID;
idURL,id_data=ReformatJSON(SummonerName,Region,Key)
SummonerID=id_data[SummonerName.lower()]["_id"]
rdata=getRecentHistory(SummonerID)
matchlist=getMatchlist(SummonerID)
```

By finding this summoner's recent games, we get the fellow players' Summoner ID in each game, and iteratively get more and more Summoner IDs.

```
# Generate an id list to store a huge amount of summoner ids
idlist = [SummonerID]

# Iterate calling the API,
# get each summoner's 10 recent games,
# find their fellow players in each game
# save their summoner id to a list, 'idlist'
for k in range(19510):
    rdata=getRecentHistory(idlist[k])
   for i in range(len(rdata["games"])):
    # In some kinds of games, player doesn't have fellow players
    if "fellowPlayers" not in rdata["games"][i].keys():
      continue
    for j in range(len(rdata["games"][i]["fellowPlayers"])):
      idlist.append(rdata["games"][i]["fellowPlayers"][j]["summonerId"])
  # Get a list of unique summoner IDs
uniidlist = list(set(idlist))
```

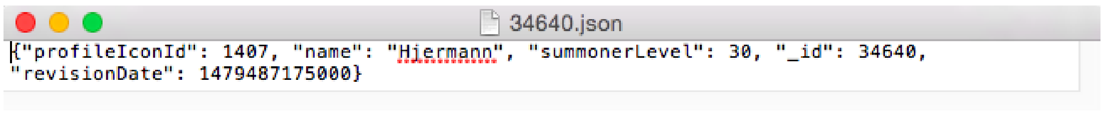


```
[29896423, 43687336, 23363880, 30477085, 44793278, 27899868, 42407249, 62622207, 38757105,
49559673, 43687336, 33471167, 30477085, 22649715, 40956892, 38690797, 38291459, 27738879,
49559673, 43687336, 19723810, 30477085, 33037964, 70577341, 42212709, 20042563, 51716726,
34628609, 25939248, 30477085, 27914729, 19004663, 48847275, 32817307, 31818413, 42648036,
41530759, 70199735, 85360582, 36764308, 47077062, 72602433, 27781162, 49290366, 28040978,
44444768, 29737467, 22901273, 25126971, 42277340, 76660449, 35257388, 28255897, 44738810,
20951688, 34627914, 72391244, 37061828, 60814394, 47249810, 85987099, 71600735, 41696364,
37933039, 21214291, 19582767, 19096814, 28306998, 20262541, 68909475, 20429674, 19132036,
35918701, 21725195, 61299975, 23386796, 27015830, 34928450, 71137057, 31227013, 23953356,
70677743, 40577640, 50536871, 20030423, 27530264, 29808703, 24870860, 24062368, 25593287,
21743792, 23363880, 30477085, 44793278, 29896423, 27899868, 42407249, 62622207, 38757105,
49559673, 33471167, 30477085, 22649715, 29896423, 40956892, 38690797, 38291459, 27738879,
49559673, 19723810, 30477085, 29896423, 33037964, 70577341, 42212709, 20042563, 51716726,
34628609, 43749164, 37904342, 24041445, 34347485, 23028635, 20893140, 31507901, 46727082,
38656834, 68458527, 48905200, 52013607, 60719927, 28314628, 83727475, 70840139, 81626895,
20393098, 52313588, 53106976, 51821656, 22211081, 43046238, 34967583, 52602437, 49559673,
```

a Quick View of the idlist File

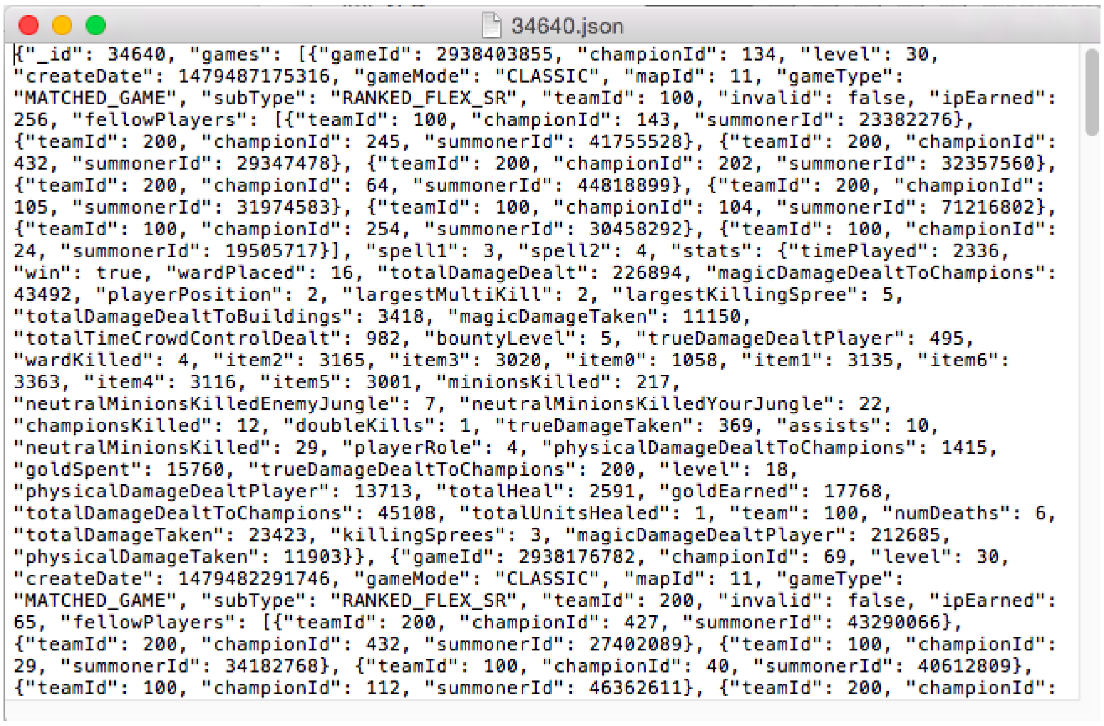
Then we stored this idlist to a local text file. By querying to elements (SummonerIds) in the list, we can get more players' recent 10 games one by one.

```
# With summoner id list, we get their 10 recent games one by one
for gamer in uniidlist:
  idURL,id_data=ReformatJSONbyid(str(gamer),Region,Key)
  # Because of the API call limit, we have to use time.sleep()
  time.sleep(0.8)
  rdata=getRecentHistory(gamer)
  time.sleep(0.8)
```


34640.json

```
{"profileIconId": 1407, "name": "Hjermann", "summonerLevel": 30, "_id": 34640,
"revisionDate": 1479487175000}
```

a Quick View of the Summoner Info File


34640.json

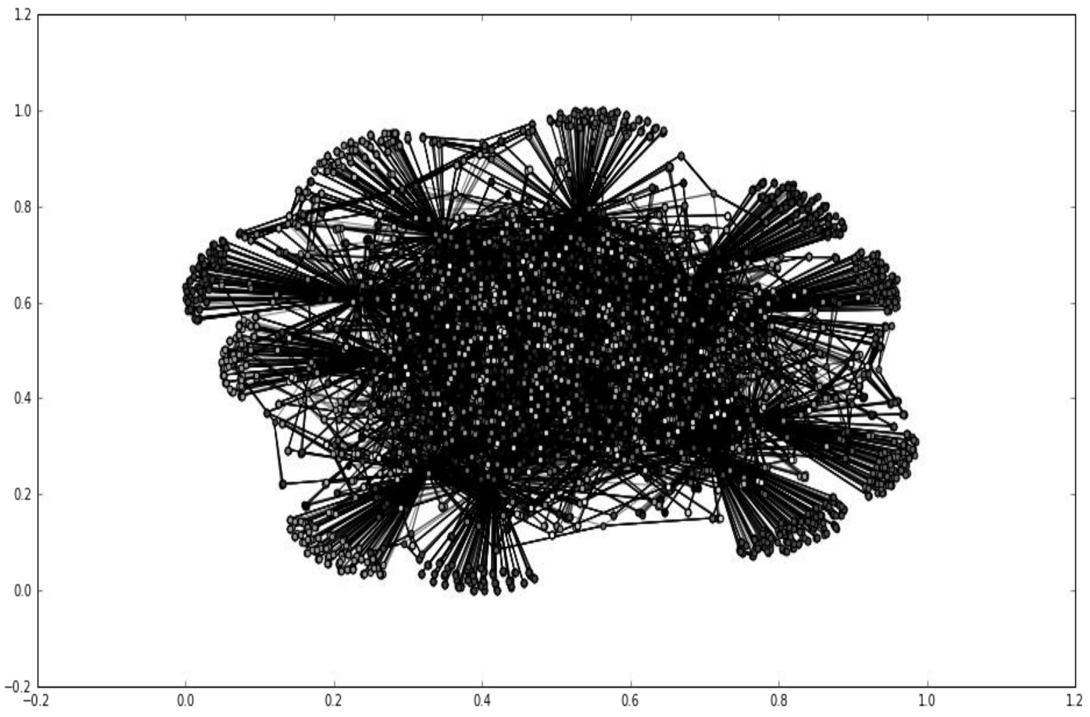
```
{"_id": 34640, "games": [{"gameId": 2938403855, "championId": 134, "level": 30,
"createDate": 1479487175316, "gameMode": "CLASSIC", "mapId": 11, "gameType":
"MATCHED_GAME", "subType": "RANKED_FLEX_SR", "teamId": 100, "invalid": false, "ipEarned":
256, "fellowPlayers": [{"teamId": 100, "championId": 143, "summonerId": 23382276},
{"teamId": 200, "championId": 245, "summonerId": 41755528}, {"teamId": 200, "championId":
432, "summonerId": 29347478}, {"teamId": 200, "championId": 202, "summonerId": 32357560},
{"teamId": 200, "championId": 64, "summonerId": 44818899}, {"teamId": 200, "championId":
105, "summonerId": 31974583}, {"teamId": 100, "championId": 104, "summonerId": 71216802},
{"teamId": 100, "championId": 254, "summonerId": 30458292}, {"teamId": 100, "championId":
24, "summonerId": 19505717}], "spell1": 3, "spell2": 4, "stats": {"timePlayed": 2336,
"win": true, "wardPlaced": 16, "totalDamageDealt": 226894, "magicDamageDealtToChampions":
43492, "playerPosition": 2, "largestMultiKill": 2, "largestKillingSpree": 5,
"totalDamageDealtToBuildings": 3418, "magicDamageTaken": 11150,
"totalTimeCrowdControlDealt": 982, "bountyLevel": 5, "trueDamageDealtPlayer": 495,
"wardKilled": 4, "item2": 3165, "item3": 3020, "item0": 1058, "item1": 3135, "item6":
3363, "item4": 3116, "item5": 3001, "minionsKilled": 217,
"neutralMinionsKilledEnemyJungle": 7, "neutralMinionsKilledYourJungle": 22,
"championsKilled": 12, "doubleKills": 1, "trueDamageTaken": 369, "assists": 10,
"neutralMinionsKilled": 29, "playerRole": 4, "physicalDamageDealtToChampions": 1415,
"goldSpent": 15760, "trueDamageDealtToChampions": 200, "level": 18,
"physicalDamageDealtPlayer": 13713, "totalHeal": 2591, "goldEarned": 17768,
"totalDamageDealtToChampions": 45108, "totalUnitsHealed": 1, "team": 100, "numDeaths": 6,
"totalDamageTaken": 23423, "killingSprees": 3, "magicDamageDealtPlayer": 212685,
"physicalDamageTaken": 11903}}, {"gameId": 2938176782, "championId": 69, "level": 30,
"createDate": 1479482291746, "gameMode": "CLASSIC", "mapId": 11, "gameType":
"MATCHED_GAME", "subType": "RANKED_FLEX_SR", "teamId": 200, "invalid": false, "ipEarned":
65, "fellowPlayers": [{"teamId": 200, "championId": 427, "summonerId": 43290066},
{"teamId": 200, "championId": 432, "summonerId": 27402089}, {"teamId": 100, "championId":
29, "summonerId": 34182768}, {"teamId": 100, "championId": 40, "summonerId": 40612809},
{"teamId": 100, "championId": 112, "summonerId": 46362611}, {"teamId": 200, "championId":
```

a Quick View of the Summoner's Recent Game History File

We also downloaded the champion mastery of each player. Now with 32,661 summoner data downloaded, 326,610 games ready to be analysed, we decide to work with these data files and see if there's anything interesting.

Later, we will store the data in MongoDB, as explained in detail in part 4, MongoDB.

It will be natural for us to think of the way we downloaded data as a social network --- some players play games together, and they can be linked to each other. Then we created a graph, linking all the players in each game in pairs. We used about 5000 summoners, otherwise it will be hard to plot the graph. The lighter color one node is, the more connected to other nodes it is. You can see the code in the Appendix.



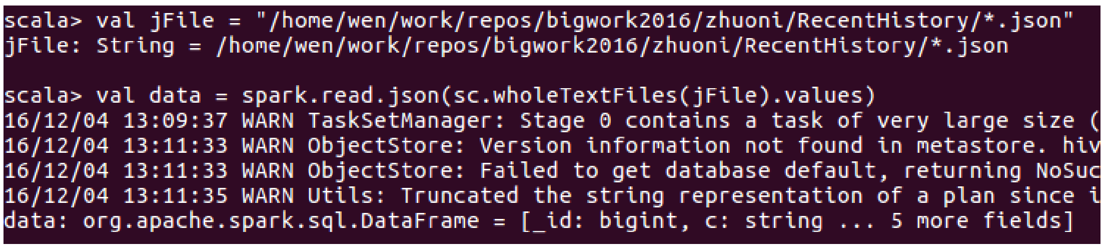
3. Spark Mapreduce

Spark Mapreduce method is used to process multiple JSON files. In the section we will explain how we use the Spark Mapreduce to filter out related data from the large datasets. We will use an example of filter out each 'championId' and it's appearance.

As we mentioned the process flow in the design phase 3, the first step is to convert the all JSON files to Spark SQL Dataframe.

```
//select all json file from a desired path
val jFile = "/PATH/*.json"

//convert all json file to spark sql dataframe
val data = spark.read.json(sc.wholeTextFiles(jFile).values)
```

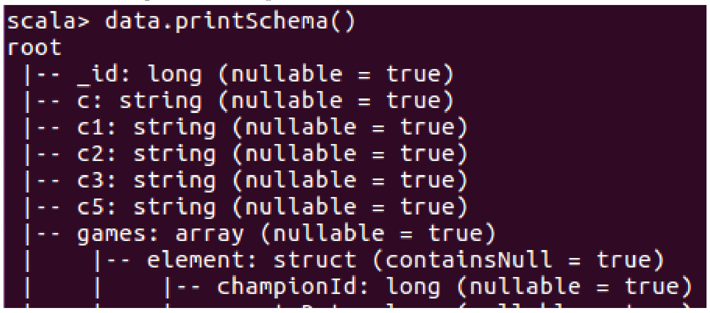
The code above is the scala command to do the conversion. Then we can check the schema of the data frame for all JSON files.



```
scala> val jFile = "/home/wen/work/repos/bigwork2016/zhuoni/RecentHistory/*.json"
jFile: String = /home/wen/work/repos/bigwork2016/zhuoni/RecentHistory/*.json

scala> val data = spark.read.json(sc.wholeTextFiles(jFile).values)
16/12/04 13:09:37 WARN TaskSetManager: Stage 0 contains a task of very large size (
16/12/04 13:11:33 WARN ObjectStore: Version information not found in metastore. hiv
16/12/04 13:11:33 WARN ObjectStore: Failed to get database default, returning NoSuc
16/12/04 13:11:35 WARN Utils: Truncated the string representation of a plan since i
data: org.apache.spark.sql.DataFrame = [_id: bigint, c: string ... 5 more fields]
```

The figure above is to show the terminal response which spark has convert the JSON file to the Spark SQL Dataframe.

```
//check the schema
            data.printSchema()
```



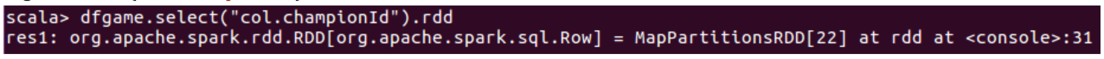
```
scala> data.printSchema()
root
 |-- _id: long (nullable = true)
 |-- c: string (nullable = true)
 |-- c1: string (nullable = true)
 |-- c2: string (nullable = true)
 |-- c3: string (nullable = true)
 |-- c5: string (nullable = true)
 |-- games: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- championId: long (nullable = true)
```

The code above is to check the schema. We can see the 'championID' is a structed low level key under the key 'games'. If we want to select this key we have to destruct it first.

```
//destruct the dataframe
val dfgame = data.select(explode(data("games")))
```

After destruct the data frame we can convert the Spark SQL Dataframe to RDD.

```
//select championId
dfgame.select("col.championID").RDD
```



```
scala> dfgame.select("col.championId").rdd
res1: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[22] at rdd at <console>:31
```

The figure above shows the the SQL data frame is saved as the RDD successfully. After that we can use mapreduce method to pair the 'championId' and a integer 1 then reduce it by count the appearance.

```
//select championId
dfgame.select("col.fellowPlayers.championID").rddrdd.map(word => (word, 1)).reduceByKey(_ +
_).saveAsTextFile("123")
```

4. MongoDB

MongoDB (from *humongous*) is a free and open-source cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schemas (source: wikipedia). Using MongoDB, we want to store our dataset, make some queries, and analyze some statistical results.

We want to explore these things:

- The overall win-rate of each champion,
- The win-rate of each champion group by game roles,
- The popularity of all the champions,
- The popularity of each champion group by game roles,
- Get some results on raw game stats:
    - Game impact: assists, number-of-deaths
    - Damage dealt: total damage, magical damage, physical damage

We already have PyMongo installed and it works correctly. The first step when working with PyMongo is to create a MongoClient to the running mongod instance. After getting a database, we got 3 collections.

```
# Connecting to mongoDB database
client = pymongo.MongoClient()
db = client['RiotMongoDB']

SummonerID_Collection = db["SummonerID"]
GamesStats_Collection = db["GamesStats"]
ChampionCount_Collection = db["ChampionCount"]
```

Then we used glob.glob() function to get all the file names in the working respiratory, read all the files we downloaded, and stored all the data we need.

For SummonerID_Collection, we have 3 keys: '_id', 'name', 'summonerLevel'.

For GamesStats_Collection, we have 2 keys: '_id', 'games'. 'Games' is a dictionary, which has many subkeys, like 'championId', 'createDate', 'fellowPlayers', 'stats', 'gameMode', etc. In 'stats', we have much more things, like 'win', 'assists', 'killingSprees', etc. These are all the raw data we need to analyze each champion's performance in games.

Then, we make some simple queries to ensure our database works.

```
# Get the names for all the current databases
db.collection_names()
```

Getting this list of all the names for current databases:

```
[u'ChampionCount', u'GamesStats', u'SummonerID']
```

```
# Get a single sample document
SummonerID_Collection.find_one()
```

Getting this summoner's information:

```
{u'_id': 102363, u'name': u'Twitch DiabloEMT', u'summonerLevel': 30}
```

Then we count all the summoners we have:

```
# Conduct a simple 'count' query
SummonerID_Collection.count("_id")
```

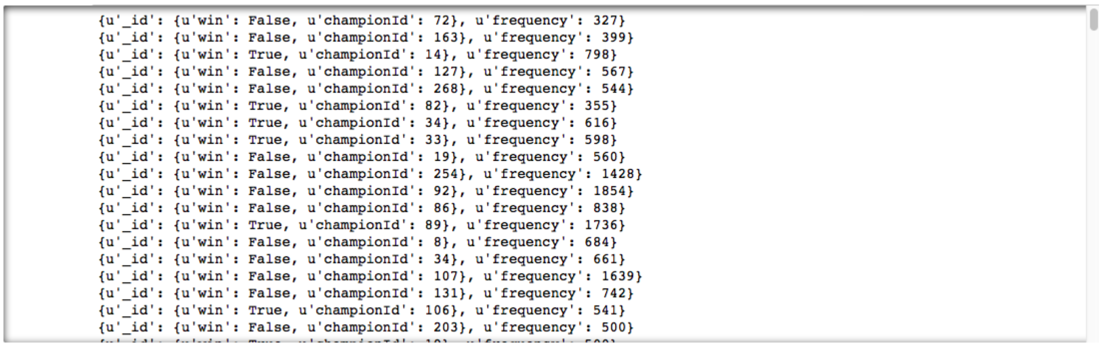
Getting a result of 32660.
After these simple queries, we counted the frequency of each champion using aggregation pipeline, group by their championId and win state. This is because we want to calculate their win-rate and popularity. Therefore, we can get how many times the champions are used by players, and how many times the champions won the games.

```
# Count the frequency of each champion using aggregation pipeline
for item in GamesStats_Collection.aggregate([{"$unwind": "$games"},
                        {"$group" : {"_id" : {'championId':"$games.championId",
                        'win':"$games.stats.win"}, "frequency" : {"$sum" : 1}}}]):
   entryID = ChampionCount_Collection.insert_one(item)
```

Fellow players are players who also attended the same game. In one game, one summoner has 9 fellow players. We also considered the fellow players' champions. The different thing is, if we want to determine if a fellow player wins, we need to find if the fellow player is in the same team as the summoner.
So we used a conditional aggregation in this case, to determine if a fellow player is in the same team as the original summoner.

```
for item in GamesStats_Collection.aggregate([{"$unwind": "$games.fellowPlayers"},
                        {"$group" : {"_id" : {'championId':"$games.fellowPlayers.championId",
                        'win': { "$cond": [ { "$eq": [ "$games.fellowPlayers.teamId", "$games.teamId" ] } ,
                        "$games.stats.win", {"$not":["$games.stats.win"]} ]}
                        }, "frequency" : {"$sum" : 1}}}]):
```

Here is a sample raw output.

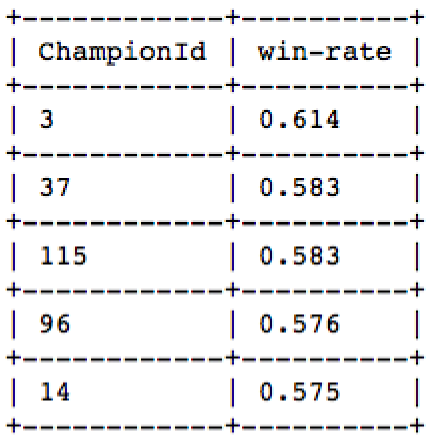


```
{u'_id': {u'win': False, u'championId': 72}, u'frequency': 327}
{u'_id': {u'win': False, u'championId': 163}, u'frequency': 399}
{u'_id': {u'win': True, u'championId': 14}, u'frequency': 798}
{u'_id': {u'win': False, u'championId': 127}, u'frequency': 567}
{u'_id': {u'win': False, u'championId': 268}, u'frequency': 544}
{u'_id': {u'win': True, u'championId': 82}, u'frequency': 355}
{u'_id': {u'win': True, u'championId': 34}, u'frequency': 616}
{u'_id': {u'win': True, u'championId': 33}, u'frequency': 598}
{u'_id': {u'win': False, u'championId': 19}, u'frequency': 560}
{u'_id': {u'win': False, u'championId': 254}, u'frequency': 1428}
{u'_id': {u'win': False, u'championId': 92}, u'frequency': 1854}
{u'_id': {u'win': False, u'championId': 86}, u'frequency': 838}
{u'_id': {u'win': True, u'championId': 89}, u'frequency': 1736}
{u'_id': {u'win': False, u'championId': 8}, u'frequency': 684}
{u'_id': {u'win': False, u'championId': 34}, u'frequency': 661}
{u'_id': {u'win': False, u'championId': 107}, u'frequency': 1639}
{u'_id': {u'win': False, u'championId': 131}, u'frequency': 742}
{u'_id': {u'win': True, u'championId': 106}, u'frequency': 541}
{u'_id': {u'win': False, u'championId': 203}, u'frequency': 500}
```

Then we are able to calculate the win-rate of each champion. If we add the win frequency and lose frequency, we can get the total frequency that a champion is used. And if we divide the win frequency by the total frequency, we can get a champion’s win rate.

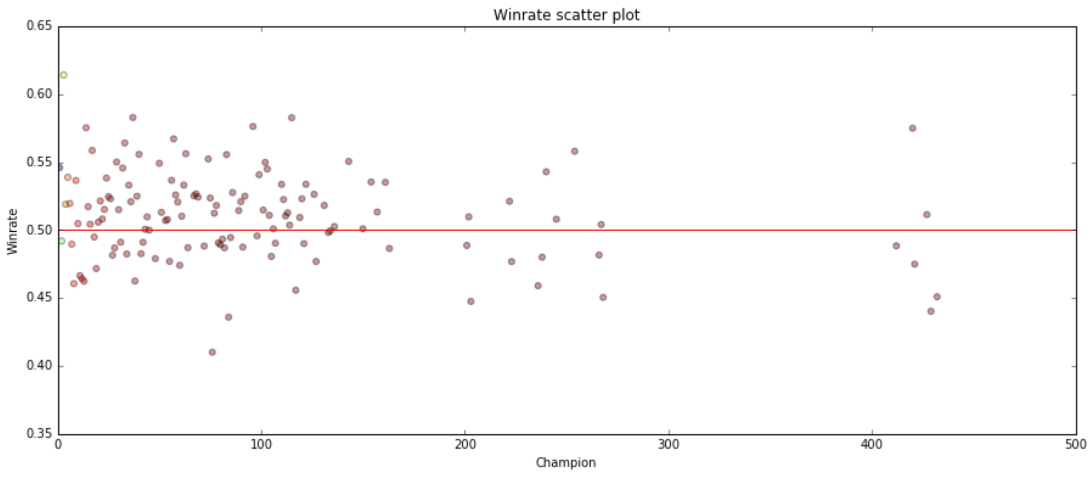
```
# Calculate the win-rate of each champion
winrate = {}
popularity = {}
for item in ChampionCount_Collection.aggregate([{"$unwind": "$_id"},
                        {"$group": {"_id": '$_id.championId',
                         'total': {"$sum": "$frequency"},
                         'win': {"$sum": { "$cond": [ { "$eq": [ "$_id.win", True ] } ,
                         "$frequency", 0 ]}}}}]):
    Id = item['_id']
    winrate.update({Id:item['win']/item['total']})
    popularity.update({Id:item['total']})
```

And we got the top 5 champions with highest win-rate in a table.



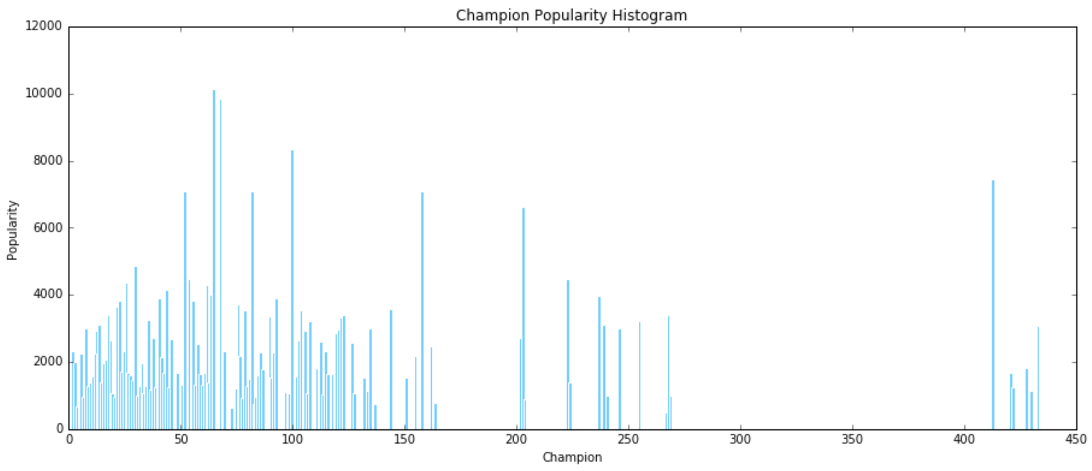
```
+------------+----------+
| ChampionId | win-rate |
+------------+----------+
| 3          | 0.614    |
+------------+----------+
| 37         | 0.583    |
+------------+----------+
| 115        | 0.583    |
+------------+----------+
| 96         | 0.576    |
+------------+----------+
| 14         | 0.575    |
+------------+----------+
```

To see how the win-rate of all the champions distributes, we did a scatter plot, with x-axis being the championId (note that the champion Id are not consecutive), and y-axis being the win-rate.



We can see the win-rate fluctuates between 0.40 and 0.65, with an approximate mean value of 0.50. It is very dense at around the 0.50 line. This is reasonable, because we expect that each champion has a win-rate of about 0.50, but not so different from 0.50.

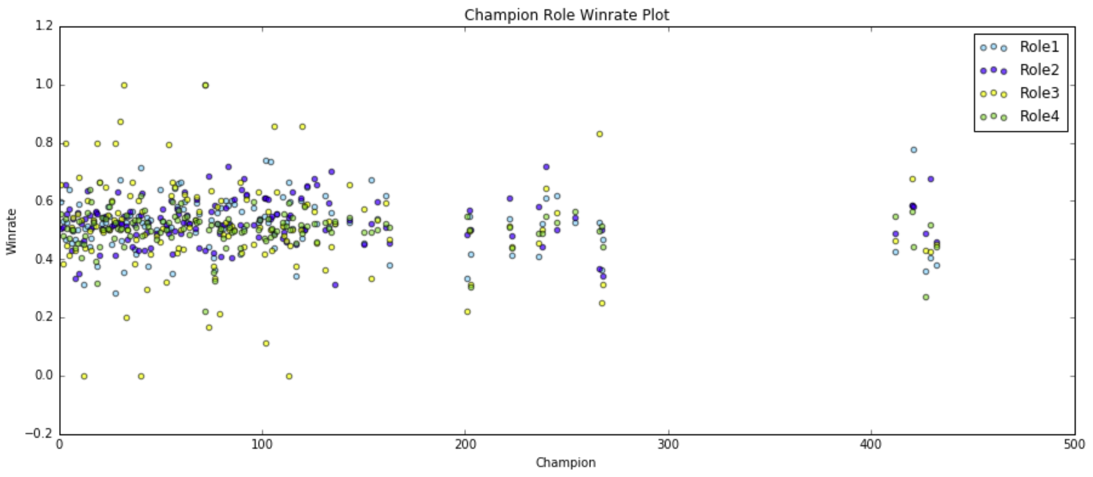
Besides, we can also find that some champions such as 3# have higher win-rate. According to our later analysis, champions who have higher win-rate are not necessarily more popular than others. Below is another plot of the champions' popularities.



We can see that several champions are far more popular than the others. And several champions are very rarely used.

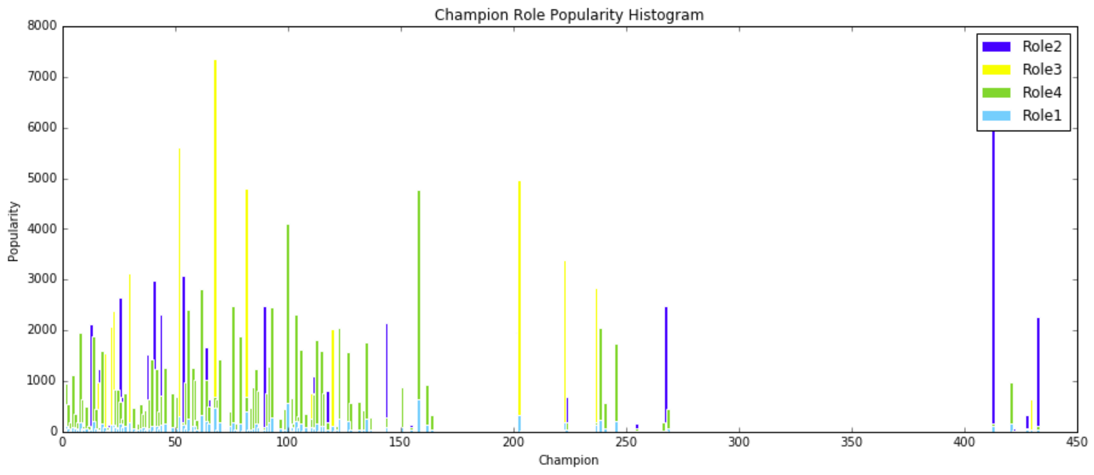
In a game, one player has 4 roles. Player role is a key in 'stats', the value can be 1, 2, 3 or 4 (DUO(1), SUPPORT(2), CARRY(3), SOLO(4)). Next, we want to find for each role, how are the champions' win-rates distributed, and which champion is more popular.

To get the desired results, we created a list of dictionaries to store each role's champion win-rate data and champion popularity (frequency) data. We first got this overall scatter plot.



We can see some champions have odd high win-rates. This is because these champions are very rarely played on this role, therefore lacking statistics to get the more precise win-rate.

Generally, champions playing role 4 have lower win-rate. And the win-rate of champions playing role 5 has a big variance, which means it can be very different for different champions. Again, all the champions' win-rate are at around 0.5.



Then we plotted the histogram of champions' popularities. It is clear seen from the histogram that role 1 is much rarer played by summoners. Besides, several champions are extremely popular for role 3. And we can see from the scatter plot above that they have relatively high win-rates. Only several champions are chosen to play role 2. And the popularity of champions playing role 4 doesn't vary too much.

Next, we use aggregate in MongoDB to consider some statics in games, and sort them by value to see which champions have the highest capability in this specific field.

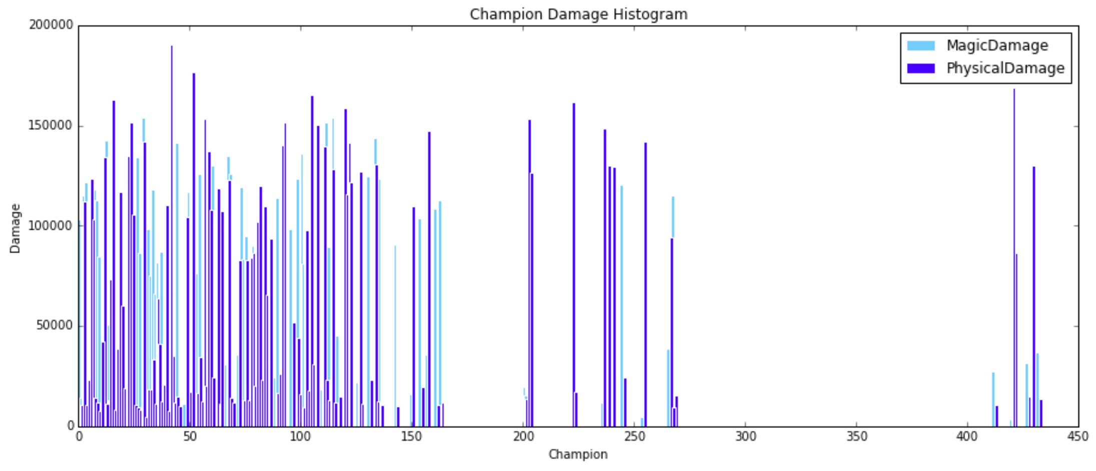
```
for item in GamesStats_Collection.aggregate([{"$unwind": "$games"},
                                    {"$group": {"_id": '$games.championId',
                                     'avgassists': {"$avg": "$games.stats.assists"}}},
                                    {"$sort": {'avgassists': -1}}]):
  t.add_row([item['_id'], item['avgassists']])
  assist.update({item['_id']:item['avgassists']})
```

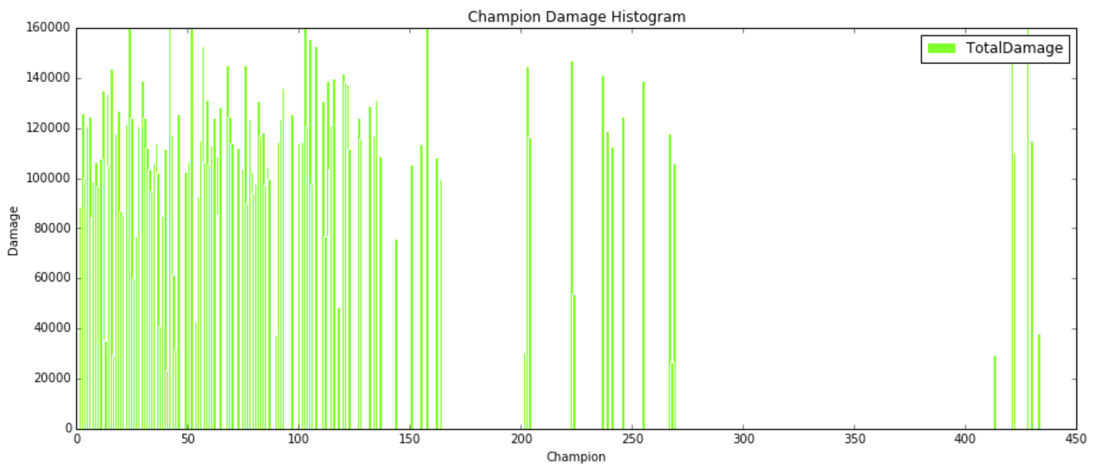
This sample code is used to getting the average assists of each champion group by championId, and sort them in descending order. Although we chose to store the result in a dictionary, which has no order attributes, we also stored the result one by one in a table, so we can still get the sorted results. We did this similarly for number-of-deaths. And here is a sample output:

```
+------------+------------+  +------------+-----------+
| ChampionId | avgassists |  | ChampionId | avgdeaths |
+------------+------------+  +------------+-----------+
| 3          | 21.139     |  | 40         | 5.167     |
+------------+------------+  +------------+-----------+
| 37         | 20.588     |  | 154        | 5.822     |
+------------+------------+  +------------+-----------+
| 16         | 20.295     |  | 427        | 5.912     |
+------------+------------+  +------------+-----------+
| 44         | 20.187     |  | 267        | 6.031     |
+------------+------------+  +------------+-----------+
| 267        | 19.325     |  | 102        | 6.190     |
+------------+------------+  +------------+-----------+
| 40         | 19.182     |  | 78         | 6.195     |
+------------+------------+  +------------+-----------+
| 201        | 18.040     |  | 16         | 6.263     |
+------------+------------+  +------------+-----------+
| 20         | 17.864     |  | 98         | 6.352     |
+------------+------------+  +------------+-----------+
```

Using the similar method, we consider different kinds of damage that champions do in games. In lol games, the most important damages are total damage, magic damage, and physical damage.

And we got the following 2 histograms.





From the magic/physical damage histogram, we can see that champions with higher magic damage usually have lower physical damage. This makes sense, because the champions should not differ too much in overall capabilities.

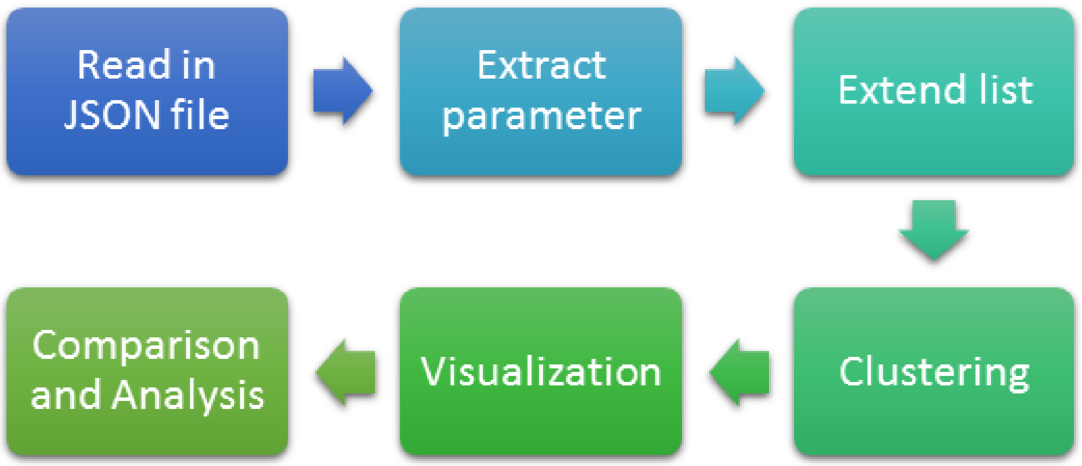
As for the total damage histogram, we can see that apart from some certain champions, most champions have similar total damages. And as for these champions who have much lower total damage, they may have other special advantages.

5.DBSCAN Clustering Analysis

Perhaps a most exciting feature of the Riot LoL is the abundant classes and roles in the game. In each game, the summoner would choose his/her own champion at the beginning, and in most cases the champions in a team would be of different classes. Generally speaking, all champions belong to six main classes, for example, Fighter, Tank and Mage. Champions of the same class usually have similar attributes and skills, while some distinctions do exist. And interestingly, a champion sometimes belong to more than one classes.

Maybe you have learned that in LoL the major purpose of a player is to attack and even kill enemies, and two kinds of impact could be exacted on rivals are Magic Damage and Physical Damage. Considering the classes and champions that we have mentioned before, we are naturally curious about the performance of champions belonging to distinctive classes. Is a certain champion better at Magic Damage rather than Physical Damage? Do summoners choosing the same champion have similar performance? And do champions of the same class share some properties? For finding the keys to questions above, a cluster analysis by DBSCAN method would be a good choice.

5.1. Structure of Code



Flow Chart of DBSCAN Clustering Analysis

In this part of DBSCAN, most codes are encapsulated into three functions, *GetPoint()*, *ReadFile()* and *DBscan()*. Here are the introduction of these three functions.

GetPoint(Path, stats_x, stats_y) is for extraction of data required from a certain JSON file. Among three parameters, *Path* indicates the direction of a certain JSON file, while *stats_x* and *stats_y* are names of two statistics. On receiving the objective direction, this function would load all the content from the file first. Then for record of each match, it would extract the values of two given keys. Of course there is a judgment to ensure that such values exist. At last it would return expected list of parameters.

ReadFile(Folder, Objective_ChampionId, stats_x, stats_y) is meant for dealing with all JSON files. It would open the folder containing all the JSON files first and utilize *GetPoint()* to extract the data one by one. Since we are only interested in some champions, a judgment is used to abandon the data we do not concern. Finally we would receive a numpy array of points with objective Champion ID.

DBscan(dataset, Eps, Min_Samples) is the key function of cluster analysis, serving for both clustering and visualization. Two of the parameters, *Eps* and *Min_Samples*, determine the quality of clustering and have been mentioned in the course. And the other parameter *dataset* is the outcome of *ReadFile()*. After the process of clustering, the package *matplotlib.pyplot* is used to plot the result.

Main part of the code is like this

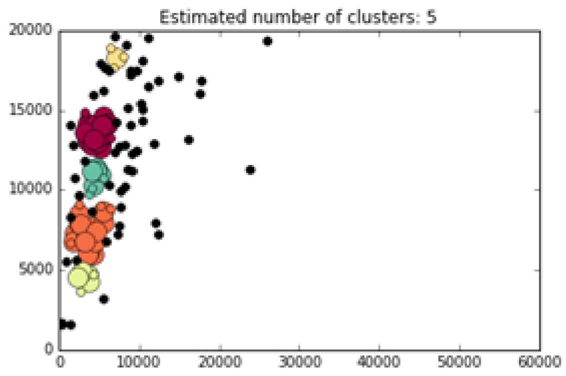
```
champion = [103, 55, 14, 117, 107, 16, 77, 127, 91, 3, 236, 10, 22, 68, 268, 44]
for i in champion:
  dataset = ReadFile('RecentHistory', i, 'magicDamageDealtToChampions',
'physicalDamageDealtToChampions')
  DBscan(dataset, 1250, 6)
  plt.savefig('image/' + str(i) + '.png')
  plt.show()
```

Here *champion* is a list of Champion IDs, which come from different classes. And a for loop is meant to deal with each champion. The variable *dataset* receives all the value of parameters '*magicDamageDealtToChampions*' and '*physicalDamageDealtToChampions*' of champion in the folder '*RecentHistory*' with a certain champion Id. *Dbscan()* realize clustering and two other parameters could be changed. Then two lines of code respectively save and show the plot.

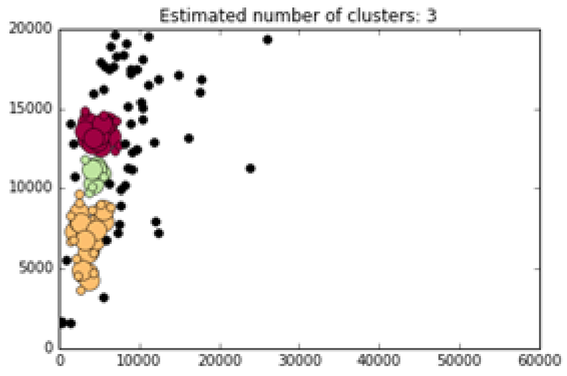
All codes are attached to Appendix.

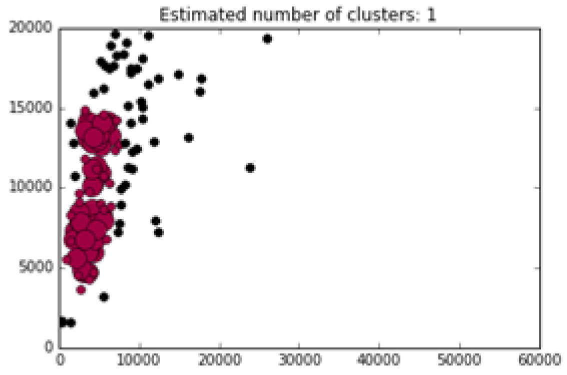
5.2. Clustering Parameter selection

Perhaps all the clustering analysis would face the problem of parameter selection. And maybe the best solution is to do experiments. From a small-range observation we find that Magic Damage usually range from 0 to 70000 while Physical Damage from 0 to 25000. Hence Eps should be around 1000-1500. Correspondingly, Min_Samples could be about 5. As a result, the combination of (1000, 4), (1250, 6), (1500, 8) are taken into consideration. We perform DBSCAN with these three kinds of parameters on a relatively smaller dataset and the result is as below



.

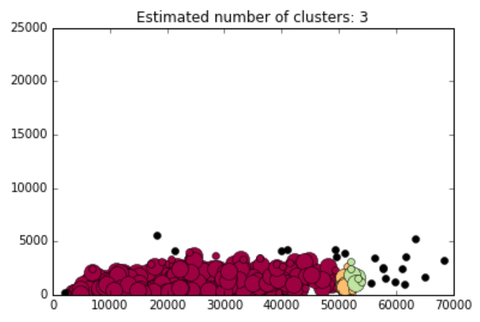
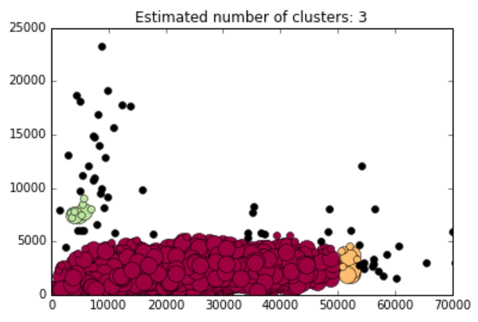
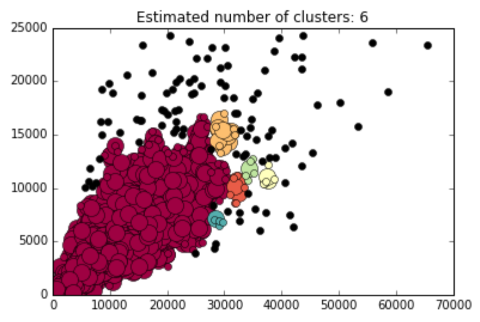
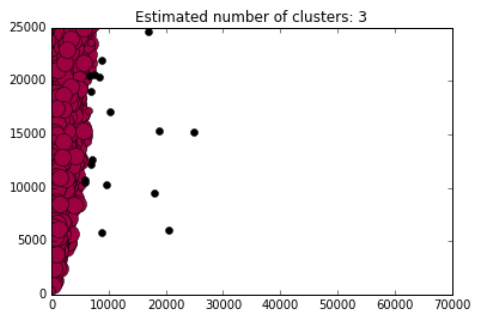




It is apparent that parameter combination (1250, 6) has the best clustering effect, since it could lead to a moderate number of clusters while correctly distinguishing neighboring clusters.

5.3. Outcomes and Analysis

Now we can perform DBSCAN clustering. But before that we still need to choose a set of champions of different classes. And the fact that one champion usually has duplicate classes make things more complicated. Eventually we choose 16 champions whose champion IDs are respectively 3, 127, 103, 55, 117, 16, 10, 77, 14, 268, 68, 44, 14, 107, 236, 22, 91. All the clustering results are attached into Appendix and four most typical ones are shown here.

	
Champion ID: 3	**Champion ID**: 103
Class: Tank, Mage	**Class**: Mage, Assassin
	
Champion ID: 10	**Champion ID**: 22
Class: Fighter, Support	**Class**: Marksman, Support

With these outcomes we safely reach some interesting outcomes. For example, for most champions a preference between Magic Damage and Physical Damage. As you can see, in each plot the points are not randomly distributed. On the contrary, the majority of points form a huge cluster, which implies that in most matches a certain champion performs in the same way.

And just as we expected, champions of exactly same classes usually perform similarly. A good example is champion No. 117 and No. 16, both of whom are Support and Mage. These two champions has the same complicated distribution. The biggest cluster appears at the lower left corner of the plot, indicating most summoners using these champions exact low/moderate Magic Damage as well as low Physical Damage on enemies. However, the rest small cluster(s) and lots of outsiders at the higher left corner shows that summoners that like low Magic Damage and high Physical Damage could not be ignored. The similar situations exist between No. 107 and No. 91, No. 103 and No. 55.

As for the single class, we could not now determine its direct relation with output preference. In other words, we could not say if Fighter preference Magic or Physical Damage. Luckily, we find some other interesting conclusion. For a Tank, Fighter, Assassin or Marksman, the champion tends to have strong preference, while a Mage or Support would be more neutral. Besides, a Fighter seldom has high Magic Damage while a Tank usually has low Physical Damage.

Conclusion

In the past four weeks we have finished the job that we never imagined before. For the first time we find and retrieve gigabyte level of data, store it, analyze it, visualize it and show it to you. At the beginning of this project, we describe it as the combination of three problems: data retrieving, data processing and data visualization. Objectively speaking, we did a good job.

Retrieving data we need is a hard job since this course did not talk much about that. And this part did take more time than we thought. Luckily we soon learned about APIs. And in this project we managed to download data through API, especially different types of data through a set of APIs.

Data processing is the main part of this course, as well as our project. In general we have utilized three tools/skills: Map Reduce by Apache Spark, MongoDB and DBSCAN clustering. Though they have appeared in previous lectures and exercises, it is in this project that they are used so frequently and to dealt with such large amount of data. The project does help us understand them more deeply.

Throughout the project, data visualization is always a vital matter, and the main tool we used is the python library *matplotlib.pyplot*. We believe that through scatters, charts and histograms the reviewer could have a better view of the project.

Appendix I - Game related words Definitions

summoner - user

summonerID - user ID given by RIOT, it is different with login ID, the user ID used as parameters to query various RIOT API

game - a game consists 10 summoners

champion - character created by RIOT, summoner select a character to play in the game

championID - champion identification is given by RIOT, to distinguish individual champion they have.

damage - damage dealt by the champion which played by certain summoner in the game

Appendix II -Riot class

```
#!/usr/bin/python

import json
import riotAPI as const
from urllib2 import urlopen

class Riot:

        ##class constructor
        def __init__(self, key, region):
                self.key = key
                self.region = region

        ##basic request
        def request(self, api):
                url = const.API['base'].format(R=self.region, A=api, K=self.key)
                r = urlopen(url)
                jobj = r.read()
                pobj = json.loads(jobj)
                return pobj

        ### get challenger league summoner info
        def apiChallenger(self, api):
                self.api = api.format(V=const.API_VERSION['league'],T=const.MATCH_TYPE['1'])
                return self.request(self.api)

        def apiMaster(self, api):
                self.api = api.format(V=const.API_VERSION['league'],T=const.MATCH_TYPE['1'])
                return self.request(self.api)

        def apiGame(self, api, ID):
                self.ID = ID
                self.api = api.format(V=const.API_VERSION['game'],ID=self.ID)
                return self.request(self.api)

        def api(self, api, ID):
                self.ID = ID
                self.api = api.format(V=const.API_VERSION['game'],ID=self.ID)
                return self.request(self.api)
```

Appendix III - Riot Constant

```
#!/usr/bin/python
KEY = {
            'wen': 'RGAPI-62c12c6d-5a21-4d8e-96b2-c3cad4aa9baa'
}

REGION = {
            'europe_nordic_east': 'eune',
            'europe_west': 'euw',
            'japan' : 'jp',
            'korea' : 'kr',
            'north_america' : 'na',
}

MATCH_TYPE = {
            '1': 'RANKED_SOLO_5x5'
}

API_VERSION = {
  'champion': 1.2,
  'current-game': 1.0,
  'featured-games': 1.0,
  'game': 1.3,
  'league': 2.5,
  'lol-static-data': 1.2,
  'lol-status': 1.0,
  'match': 2.2,
  'matchlist': 2.2,
  'stats': 1.3,
  'summoner': 1.4,
  'team': 2.4
}

API = {
            'base' : 'https://{R}.api.pvp.net/api/lol/{R}/{A}&api_key={K}',
            'challenger_league': 'v{V}/league/challenger?type={T}',
            'master_league': 'v{V}/league/master?type={T}',
            'game':    'v{V}/game/by-summoner/{ID}recent?',
            'champion': 'v{V/champion?}'
}
```

Appendix IV - Spark Map Reduce

```
//select all json file from a desired path
val jFile = "/PATH/*.json"

//convert all json file to spark sql dataframe
val data = spark.read.json(sc.wholeTextFiles(jFile).values)


//check the schema
            data.printSchema()

//destruct the dataframe
val dfgame = data.select(explode(data("games")))

//select championId
dfgame.select("col.championID").RDD


//select championId
dfgame.select("col.championID").rddrdd.map(word => (word, 1)).reduceByKey(_ + _).saveAsTextFile("123")
```

Appendix V - Code of Downloading Data

```
# Implement all the libraries we need

%matplotlib inline
import re
import io
import urllib2
import networkx as nx
import json
import glob
import numpy as np
import operator
import matplotlib.pyplot as plt
import time
import networkx as nx
import os.path
from __future__ import division
from pprint import pprint
import community

# Get JSON reply using the URL
def getJSONReply(URL):
  response = urllib2.urlopen(URL);
  html = response.read();
  data = json.loads(html);
  return data;

# Using the game-v1.3 API to get each player's recent history
def getRecentHistory(SummonerID):
    rURL= "https://" +Region.lower()+ ".api.pvp.net/api/lol/" + \
    Region.lower()+ "/v1.3/game/by-summoner/" + SummonerID+ "/recent?api_key=" + Key;
    r_data=getJSONReply(rURL);
    r_data['_id']=r_data['summonerId'];
    r_data.pop('summonerId');
    # Write the data to a local json file, naming it after this Summoner's ID
    with io.open('RecentHistory/%s.json' % str(SummonerID), 'w', encoding='utf-8') as f:
      f.write(unicode(json.dumps(r_data, ensure_ascii=False)))
    return r_data;

# Using the championmastery API to get each player's mastery of the champions
def getChampion(SummonerID):
    rURL= "https://" +Region.lower()+ ".api.pvp.net/championmastery/location/"\
    + Region+ "1/player/" + `SummonerID`+ "/champions?api_key=" + Key
    r_data=getJSONReply(rURL)
    # Write the data to a local json file, naming it after this Summoner's ID
    with io.open('Champion/%s.json' % str(SummonerID), 'w', encoding='utf-8') as f:
       f.write(unicode(json.dumps(r_data, ensure_ascii=False)))
    return r_data

# Using the summoner-v1.4 API to get each player's identity information by their names
```

```
def ReformatJSON(SummonerName,Region,Key):
   idURL = "https://" +Region.lower()+ ".api.pvp.net/api/lol/" +Region.lower()+ "/v1.4/summoner/by-name/" +
SummonerName+ "?api_key=" + Key
   id_data = getJSONReply(idURL)
   idRes=id_data[SummonerName.lower()]
   idRes['_id'] = idRes['id']
   idRes.pop('id')
   return idRes,id_data

# Using the summoner-v1.4 API to get each player's identity information by their IDs
def ReformatJSONbyid(SummonerID,Region,Key):
   idURL = "https://" +Region.lower()+ ".api.pvp.net/api/lol/" +Region.lower()+ "/v1.4/summoner/" +SummonerID+
"?api_key=" +Key
   id_data = getJSONReply(idURL)
   idRes=id_data[SummonerID]
   idRes['_id'] = idRes['id']
   idRes.pop('id')
   # Write the data to a local json file, naming it after this Summoner's ID
   with io.open('Summoner/%s.json' % str(SummonerID), 'w', encoding='utf-8') as f:
       f.write(unicode(json.dumps(idRes, ensure_ascii=False)))
   return idRes,id_data

SummonerName="Miladena"
Region="EUW"
Key="RGAPI-20253dda-c325-4a7e-947a-d283af4f8641"

# Retrieving the SummonerID;
idURL,id_data=ReformatJSON(SummonerName,Region,Key)

SummonerID=id_data[SummonerName.lower()]["_id"]
rdata=getRecentHistory(str(SummonerID))

# Generate an id list to store a huge amount of summoner ids
idlist = [SummonerID]

# Iterate calling the API,
# get each summoner's 10 recent games,
# find their fellow players in each game
# save their summoner id to a list, 'idlist'
for k in range(10):
   rdata=getRecentHistory(idlist[k])
   for i in range(len(rdata["games"])):
      # In some kinds of games, player doesn't have fellow players
      if "fellowPlayers" not in rdata["games"][i].keys():
         continue
      for j in range(len(rdata["games"][i]["fellowPlayers"])):
         idlist.append(rdata["games"][i]["fellowPlayers"][j]["summonerId"])

print len(set(idlist))

#========iterate untill we get sufficiently large amount of summoner ids=========
uniidlist = list(set(uniidlist))

i=0
# With summoner id list, we get their 10 recent games one by one
for gamer in uniidlist:
    idURL,id_data=ReformatJSONbyid(str(gamer),Region,Key)
    # Because of the API call limit, we have to use time.sleep()
    time.sleep(0.8)
    rdata=getRecentHistory(gamer)
    i += 1
    print i
    time.sleep(0.8)


#=========plot the graph===========

%matplotlib inline
import networkx as nx
import operator
import matplotlib.pyplot as plt
import community
from os import listdir

filename = listdir("RecentHistory")[1:]

# Creat a list of lists "players",
# store all the players' id in one game into one sublist,
# and then store the sublists to the list
players = []
for i in range(2000):
    with open('RecentHistory/%s' % filename[i]) as f:
        data = json.load(f)

    for i in range(len(data["games"])):
        if "fellowPlayers" not in data["games"][i].keys():
            continue
        # Filename is like "summonerid.json", so we use [:-5] to extract '.json'
        temp = [int(filename[i][:-5])]
        # Append the fellow players to the sublist
        for j in range(len(data["games"][i]["fellowPlayers"])):
            temp.append(data["games"][i]["fellowPlayers"][j]["summonerId"])
        # Append the sublist to the total list
        players.append(temp)

# Create a graph using networkx library
G=nx.Graph()
for j in range(1000):
    for i in range(len(players[j])-1):
        for k in range(i+1,len(players[j])):
            G.add_edge(players[j][i],players[j][k])

# Set the plot figure size
```

```
plt.figure(figsize=(16,8))

# first compute the best partition
partition = community.best_partition(G)
size = float(len(set(partition.values())))
pos = nx.spring_layout(G)
count = 0.
for com in set(partition.values()) :
    count = count + 1.
    list_nodes = [nodes for nodes in partition.keys()
                                if partition[nodes] == com]
    nx.draw_networkx_nodes(G, pos, list_nodes, node_size = 20,
                                node_color = str(count / size))

nx.draw_networkx_edges(G, pos, alpha=0.5)
plt.show()
```

Appendix VI - Code of MongoDB

```
%matplotlib inline
import io
import json
import glob
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import pyplot
import pymongo
from pymongo import MongoClient
from __future__ import division
from pprint import pprint
import operator
from texttable import Texttable

# Connecting to mongoDB database
client = pymongo.MongoClient()
db = client['RiotMongoDB']

SummonerID_Collection = db["SummonerID"]
GamesStats_Collection = db["GamesStats"]
ChampionMastery_Collection = db["ChampionMastery"]
ChampionCount_Collection = db["ChampionCount"]

sumdir = glob.glob("Summoner/*.json")

for item in sumdir:
    with io.open(item , 'r', encoding='utf-8') as json_data:
        summoner = json.load(json_data)
        json_data.close()
    entryID = SummonerID_Collection.insert_one({'_id':summoner['_id'], 'name':summoner['name'],
'summonerLevel':summoner['summonerLevel']})

sumdir = glob.glob("RecentHistory/*.json")

for item in sumdir:
    with io.open(item , 'r', encoding='utf-8') as json_data:
        history = json.load(json_data)
        json_data.close()
    for i in range(len(history['games'])):
        entryID = GamesStats_Collection.insert_one({'id':history['_id'], 'games':history['games'][i]})

# Get the names for all the current databases
db.collection_names()

# Get a single sample document
SummonerID_Collection.find_one()

# Conduct a simple 'count' query
SummonerID_Collection.count("_id")
```

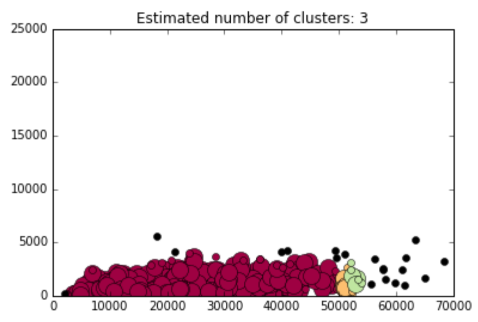
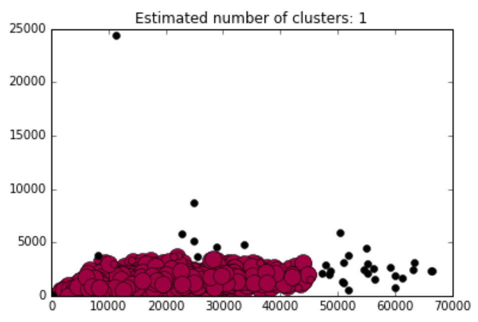
```
# Count the frequency of each champion using aggregation pipeline
for item in GamesStats_Collection.aggregate([{"$unwind": "$games"},
                                            {"$group" : {"_id" : {'championId':"$games.championId",'win':"$games.stats.win"}, "frequency" :
{"$sum" : 1}}}]):
    entryID = ChampionCount_Collection.insert_one(item)

# Count the frequency of each champion considering the summoners' fellow players
for item in GamesStats_Collection.aggregate([{"$unwind": "$games.fellowPlayers"},
                                            {"$group" : {"_id" : {'championId':"$games.fellowPlayers.championId",
                                                        'win': { "$cond": [ { "$eq": [ "$games.fellowPlayers.teamId", "$games.teamId" ] } ,
"$games.stats.win", {"$not":["$games.stats.win"]} ]}
                                                        }, "frequency" : {"$sum" : 1}}}]):
    if GamesStats_Collection.find({"_id":item["_id"]}) > 0:
        for node in GamesStats_Collection.find({"_id":item["_id"]}):
            node['frequency'] += item['frequency']
    else:
        entryID = ChampionCount_Collection.insert_one(item)

for item in ChampionCount_Collection.find():
    print item

# Calculate the win-rate of each champion
winrate = {}
popularity = {}
for item in ChampionCount_Collection.aggregate([{"$unwind": "$_id"},
                                               {"$group": {"_id": '$_id.championId',
                                                'total': {"$sum": "$frequency"},
                                                'win': {"$sum": { "$cond": [ { "$eq": [ "$_id.win", True ] } , "$frequency", 0 ]}}}}]):
    Id = item['_id']
    winrate.update({Id:item['win']/item['total']})
    popularity.update({Id:item['total']})

t = Texttable()
t.add_row(['ChampionId', 'win-rate'])

sorted_winrate = sorted(winrate.items(), key=operator.itemgetter(1), reverse=True)

for i in range(5):
    t.add_row([sorted_winrate[i][0], sorted_winrate[i][1]])

print t.draw()

plt.figure(figsize=(15,6))
T=np.arctan2(winrate.keys(),winrate.values())
plt.scatter(winrate.keys(),winrate.values(),c=T,s=25,alpha=0.4,marker='o')
plt.axhline(y=.5, xmin=0, xmax=500, linewidth=1, color = 'r')
plt.xlim([0,500])
plt.title("Winrate scatter plot")
plt.xlabel("Champion")
plt.ylabel("Winrate")
plt.show()
```

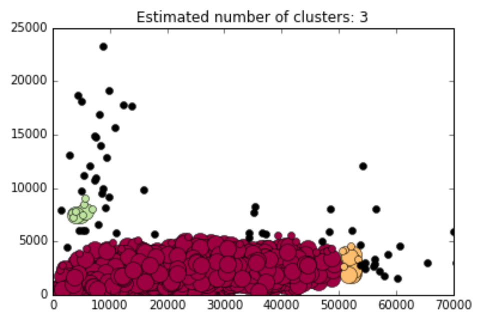
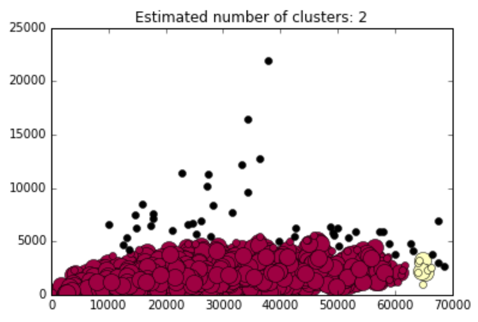
```
X = popularity.keys()
Y = popularity.values()
plt.figure(figsize=(15,6))
plt.bar(X,Y,width = 1.5,facecolor = 'lightskyblue',edgecolor = 'white')
plt.title("Champion Popularity Histogram")
plt.xlabel("Champion")
plt.ylabel("Popularity")
plt.show()

# Count the frequency of each champion using aggregation pipeline
wrolefreq = []
wrole1 = {}
wrole2 = {}
wrole3 = {}
wrole4 = {}
wrolefreq = [wrole1,wrole2,wrole3,wrole4]

lrolefreq = []
lrole1 = {}
lrole2 = {}
lrole3 = {}
lrole4 = {}
lrolefreq = [lrole1,lrole2,lrole3,lrole4]

for item in GamesStats_Collection.aggregate([{"$unwind": "$games"},
                       {"$group" : {"_id" :
{'championId':"$games.championId",'role':"$games.stats.playerRole",'win':"$games.stats.win"}, "frequency" : {"$sum" :
1}}}]):
  if "role" not in item["_id"].keys():
    continue
  if item["_id"]["win"] is True:
    wrolefreq[item["_id"]["role"]-1].update({item["_id"]["championId"]:item["frequency"]})
    continue
  if item["_id"]["win"] is False:
    lrolefreq[item["_id"]["role"]-1].update({item["_id"]["championId"]:item["frequency"]})

win = []
win1 = {}
win2 = {}
win3 = {}
win4 = {}
win = [win1,win2,win3,win4]

for i in range(4):
  for champ in wrolefreq[i].keys():
    if champ not in lrolefreq[i].keys():
      win[i].update({champ:1})
    else:
      rate = wrolefreq[i][champ]/(wrolefreq[i][champ]+lrolefreq[i][champ])
      win[i].update({champ:rate})
  for champ in lrolefreq[i].keys():
    if champ not in wrolefreq[i].keys():
        win[i].update({champ:0})

plt.figure(figsize=(15,6))
plt.scatter(win[0].keys(),win[0].values(),c = 'lightskyblue',alpha=0.6,label = 'Role1')
plt.scatter(win[1].keys(),win[1].values(),c = 'blue',alpha=0.6,label = 'Role2')
plt.scatter(win[2].keys(),win[2].values(),c = 'yellow',alpha=0.6,label = 'Role3')
plt.scatter(win[3].keys(),win[3].values(),c = 'yellowgreen',alpha=0.6,label = 'Role4')
plt.xlim([0,500])
plt.title("Champion Role Winrate Plot")
plt.xlabel("Champion")
plt.ylabel("Winrate")
plt.legend()
plt.show()

# Count the frequency of each champion group by role using aggregation pipeline
rolefreq = []
role1 = {}
role2 = {}
role3 = {}
role4 = {}
rolefreq = [role1,role2,role3,role4]

for item in GamesStats_Collection.aggregate([{"$unwind": "$games"},
                                {"$group" : {"_id" : {'championId':"$games.championId",'role':"$games.stats.playerRole"},
"frequency" : {"$sum" : 1}}}]):
  if "role" in item["_id"].keys():
    rolefreq[item["_id"]["role"]-1].update({item["_id"]["championId"]:item["frequency"]})

# Plot popularity according to roles
plt.figure(figsize=(15,6))
#plt.bar(rolefreq[0].keys(),rolefreq[0].values(),width = 1.5,facecolor = 'lightskyblue',edgecolor = 'white',label = 'Role1')
plt.bar(rolefreq[1].keys(),rolefreq[1].values(),width = 1.5,facecolor = 'blue',edgecolor = 'white',label = 'Role2')
plt.bar(rolefreq[2].keys(),rolefreq[2].values(),width = 1.5,facecolor = 'yellow',edgecolor = 'white',label = 'Role3')
plt.bar(rolefreq[3].keys(),rolefreq[3].values(),width = 1.5,facecolor = 'yellowgreen',edgecolor = 'white',label = 'Role4')
plt.bar(rolefreq[0].keys(),rolefreq[0].values(),width = 1.5,facecolor = 'lightskyblue',edgecolor = 'white',label = 'Role1')
plt.title("Champion Role Popularity Histogram")
plt.xlabel("Champion")
plt.ylabel("Popularity")
plt.legend()
plt.show()

# See the mean and variance of some statics, sort
# Game impact & damage dealt

t = Texttable()
t.add_row(['ChampionId', 'avgassists'])
assist = {}

for item in GamesStats_Collection.aggregate([{"$unwind": "$games"},
                                {"$group": {"_id": '$games.championId',
                                 'avgassists': {"$avg": "$games.stats.assists"}}},
                                {"$sort": {'avgassists': -1}}]):
   t.add_row([item['_id'], item['avgassists']])
   assist.update({item['_id']:item['avgassists']})

print t.draw()

t = Texttable()
t.add_row(['ChampionId', 'avgdeaths'])
deaths = {}
for item in GamesStats_Collection.aggregate([{"$unwind": "$games"},
                                {"$group": {"_id": '$games.championId',
                                 'avgnumDeaths': {"$avg": "$games.stats.numDeaths"}}},
                                 {"$sort": {'avgnumDeaths': 1}}]):
   t.add_row([item['_id'], item['avgnumDeaths']])
   deaths.update({item['_id']:item['avgnumDeaths']})
print t.draw()


totaldamage = {}
for item in GamesStats_Collection.aggregate([{"$unwind": "$games"},
                                {"$group": {"_id": '$games.championId',
                                 'avgtotalDamage': {"$avg": "$games.stats.totalDamageDealt"}}},
                                 {"$sort": {'avgtotalDamage': -1}}]):
   print item
   totaldamage.update({item['_id']:item['avgtotalDamage']})

for item in GamesStats_Collection.aggregate([{"$unwind": "$games"},
                                {"$group": {"_id": '$games.championId',
                                 'avgtotalDamage': {"$avg": "$games.stats.totalDamageDealt"}}},
                                 {"$sort": {'avgtotalDamage': -1}}]):
   print item

magicdamage = {}
for item in GamesStats_Collection.aggregate([{"$unwind": "$games"},
                                {"$group": {"_id": '$games.championId',
                                 'avgMagicDamage1': {"$avg": "$games.stats.magicDamageDealtToChampions"},
                                 'avgMagicDamage2': {"$avg": "$games.stats.magicDamageDealtPlayer"}}},
                                {"$project":{"_id": 1,
                                        "avgMagicDamage":{"$add":['$avgMagicDamage1','$avgMagicDamage2']}}},
                                {"$sort": {'avgMagicDamage': -1}}]):
   print item
   if item['avgMagicDamage'] is None:
      continue
   magicdamage.update({item['_id']:item['avgMagicDamage']})

physicaldamage = {}
for item in GamesStats_Collection.aggregate([{"$unwind": "$games"},
                                {"$group": {"_id": '$games.championId',
                                 'avgPhysicalDamage1': {"$avg": "$games.stats.physicalDamageDealtToChampions"},
                                 'avgPhysicalDamage2': {"$avg": "$games.stats.physicalDamageDealtPlayer"}}},
                                {"$project":{"_id": 1,
                                        "avgPhysicalDamage":{"$add":['$avgPhysicalDamage1','$avgPhysicalDamage2']}}},
                                    {"$sort": {'avgPhysicalDamage': -1}}]):
    if item['avgPhysicalDamage'] is None:
        continue
    physicaldamage.update({item['_id']:item['avgPhysicalDamage']})

plt.figure(figsize=(15,6))
plt.bar([i-1.5 for i in magicdamage.keys()],magicdamage.values(),width = 1.5,facecolor = 'lightskyblue',edgecolor =
'white',label = 'MagicDamage')
plt.bar(physicaldamage.keys(),physicaldamage.values(),width = 1.5,facecolor = 'blue',edgecolor = 'white',label =
'PhysicalDamage')
plt.xlim([0,450])
plt.title("Champion Damage Histogram")
plt.xlabel("Champion")
plt.ylabel("Damage")
plt.legend()
plt.show()

plt.figure(figsize=(15,6))
plt.bar(totaldamage.keys(),totaldamage.values(),width = 1.5,facecolor = 'greenyellow',edgecolor = 'white',label =
'TotalDamage')
plt.ylim([0,160000])
plt.title("Champion Damage Histogram")
plt.xlabel("Champion")
plt.ylabel("Damage")
plt.legend()
plt.show()

# Count the frequency of each champion using map reduce

from bson.code import Code
reducer = Code("""
            function(obj, prev){
              prev.count++;
            }
            """)

results = ChampionMastery_Collection.group(key={"championId":1}, condition={}, initial={"count": 0}, reduce=reducer)
for doc in results:
    print doc

client.close()
```

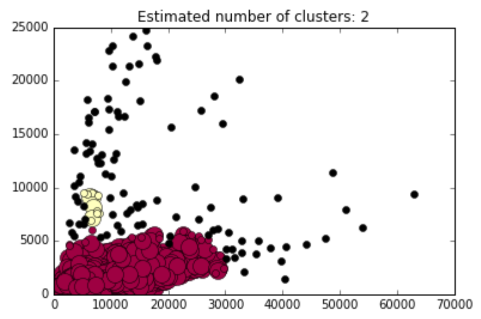
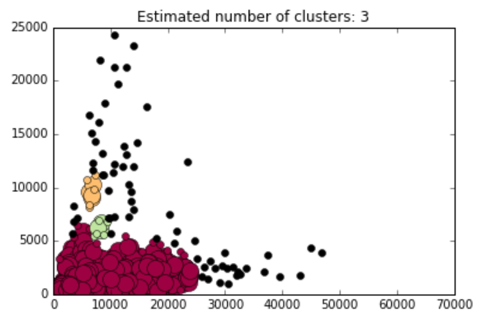
Appendix VII - Code of DBSCAN Clustering

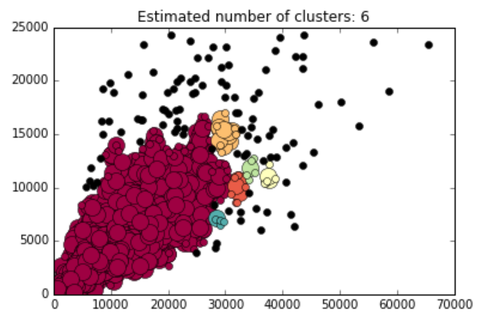
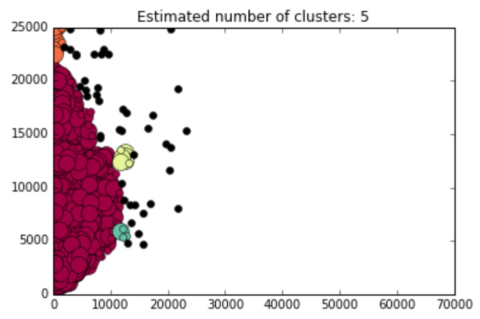
```
import numpy as np
import json
import os
from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.datasets.samples_generator import make_blobs
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

def GetPoint(Path, stats_x, stats_y):
  with open(Path) as file:
    content = file.read()
    json_content = json.loads(content)
    file.close
  games = json_content['games']
  point = []
  for i in games:
    ChampionId = i['championId']
    games_i_stats = i['stats']
    if (stats_x in games_i_stats.keys())&(stats_y in games_i_stats.keys()):
      Value = []
      Value.append(games_i_stats[stats_x])
      Value.append(games_i_stats[stats_y])
      point_i = (ChampionId, Value)
      point.append(point_i)
  return point

def ReadFile(Folder, Objective_ChampionId, stats_x, stats_y):
  Files = os.listdir(Folder)
  list_of_point = []
  for i in Files:
    dirct = Folder + '/' + i
    File_i = GetPoint(dirct, stats_x, stats_y)
    for j in File_i:
      if j[0]==Objective_ChampionId:
        list_of_point.append(j[1])
  return np.array(list_of_point)

def DBscan(dataset, Eps, Min_Samples):
  db = DBSCAN(eps=Eps, min_samples=Min_Samples).fit(dataset)
  core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
  core_samples_mask[db.core_sample_indices_] = True
  labels = db.labels_
  n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
  unique_labels = set(labels)
  colors = plt.cm.Spectral(np.linspace(0, 1, len(unique_labels)))
  for k, col in zip(unique_labels, colors):
    if k == -1:
            col = 'k'
        class_member_mask = (labels == k)
        xy = dataset[class_member_mask & core_samples_mask]
        plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=14)
        xy = dataset[class_member_mask & ~core_samples_mask]
        plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=6)
        plt.axis([0, 70000, 0, 25000])
    plt.title('Estimated number of clusters: %d' % n_clusters_)

champion = [103, 55, 14, 117, 107, 16, 77, 127, 91, 3, 236, 10, 22, 68, 268, 44]
for i in champion:
    dataset = ReadFile('RecentHistory', i, 'magicDamageDealtToChampions', 'physicalDamageDealtToChampions')
    DBscan(dataset, 1250, 6)
    plt.savefig('image/' + str(i) + '.png')
    plt.show()
```

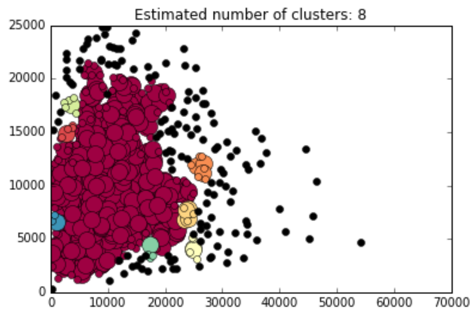
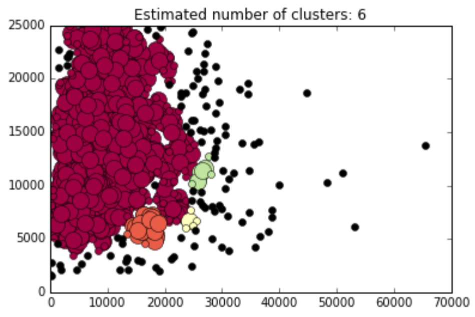
Appendix VIII - Outcomes of DBSCAN Clustering

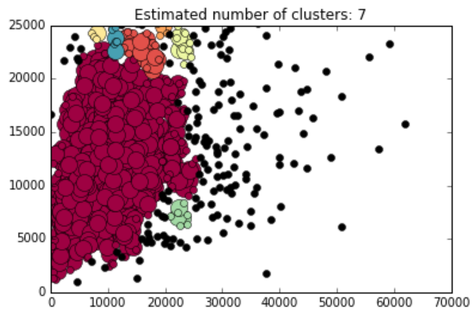
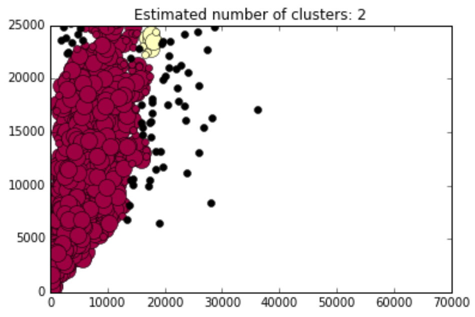
	
Champion ID: 3	**Champion ID**: 127
Name: Galio	**Name:** Lissandra
Class: Tank, Mage	**Class:** Mage

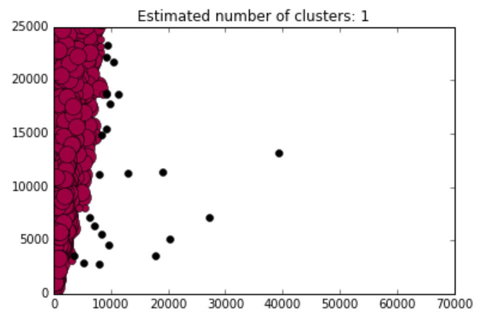
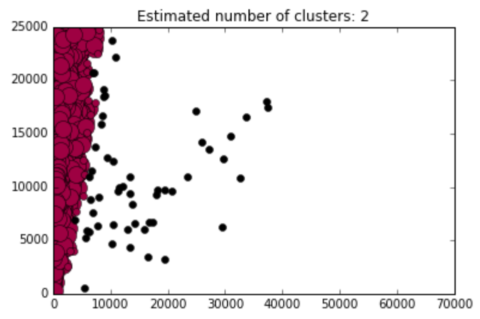
	
Champion ID: 103	**Champion ID**: 55
Name: Ahri	**Name:** Kataria
Class: Mage, Assassin	**Class:** Assassin, Mage

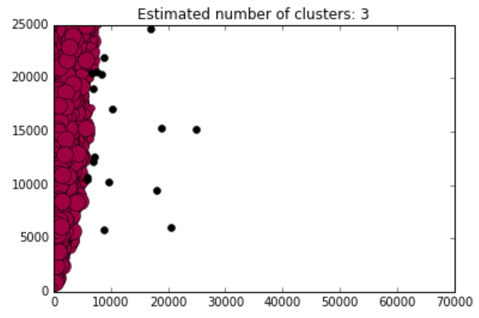
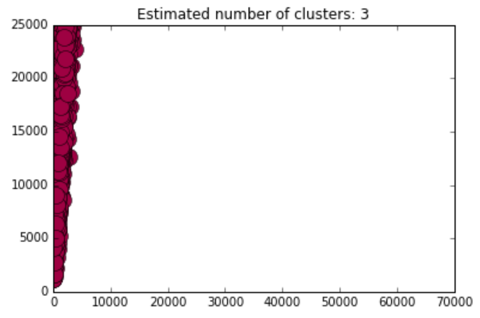
	
Champion ID: 117	**Champion ID**: 16
Name: Lulu	**Name:** Soraka
Class: Support, Mage	**Class:** Support, Mage

	
Champion ID: 10	**Champion ID**: 77
Name: Kayle	**Name:** Udyr
Class: Fighter, Support	**Class:** Fighter, Tank

	
Champion ID: 268	**Champion ID**: 68
Name: Azir	**Name:** Rumble
Class: Mage, Marksman	**Class:** Fighter, Mage

	
Champion ID: 44	**Champion ID**: 14
Name: Taric	**Name:** Sion
Class: Support, Fighter	**Class:** Tank, Fighter

	
Champion ID: 107	**Champion ID**: 236
Name: Rengar	**Name:** Lucian
Class: Assassin, Fighter	**Class:** Marksman

	
Champion ID: 22	**Champion ID**: 91
Name: Ashe	**Name:** Talon
Class: Marksman, Support	**Class:** Assassin, Fighter