BEHAVIORAL PATTERN RECOGNITION OF MULTIPLAYER ONLINE ROLE-PLAYING GAME PLAYERS USING BIG DATA ANALYTICS AND MACHINE LEARNING

FINAL REPORT

TEAM MEMBERS:

- SULEKHA ALOORRAVI
- DEEPA VENUGOPAL
- NIHARIKA THANAVARAPU
- LAKSHMIPRIYA M

MENTOR:

DR. NARAYANA D

CONTENTS

I. DOMAIN AND CONTEXT

1. Domain

A massively multiplayer online game (more commonly, MMO) is an online game which is capable of supporting large numbers of players, typically from hundreds to thousands, simultaneously from around the world.

These games can be found for most network-capable platforms, including the personal computer, video game console, or smartphones and other mobile devices. MMOs can enable players to cooperate and compete with each other on a large scale, and sometimes to interact meaningfully with people around the world.

2. Industry worth

The UK MMO-market is worth £195 million in 2009 compared to the £165 million and £145 million spent by German and French online gamers. The US gamers spend more, however, spending about $3.8 billion overall on MMO games. $1.8 billion of that money is spent on monthly subscription fees. The money spent averages out to $15.10 between both subscription and free-to-play MMO gamers. The study published by "*Today's Gamers MMO Focus Report*" also found that 46% of 46 million players in the US pay real money to play MMO games.

3. Context of this Project

It is challenging to develop the database engines that are needed to run a successful MMOG with millions of players. Understanding the behavior of players using their activity data is more important for these game developers to come up with better strategies in game development.

The variety, volume, velocity, value and veracity (Big Data 5Vs) of data that is involved in these Gaming environments exceed the limits of analysis and manipulation of conventional tools, therefore, Big Data platforms are required to handle and interpret this data.

Great volumes of data are generated all the time in these environments. Each interaction made by a player creates data that are transferred and stored, and if properly analyzed, can contain valuable information. This information can be vital for the continuity and improvement of a game. Patterns can be detected from these data and even predictive analysis can be made to foresee the actions and intentions of the players inside the game.

4. Objective

Objective of this Project is to perform analytics on one such Big Data Gaming Environment and the results would help game developers in:

- Optimizing user experience
- Improving revenue
- Raise the level of control over the environment

II. SUMMARY

2.1 Problem Statement

2.1.1 PERFORM EXPLORATORY ANALYSIS

1.1 To cluster players into different groups based on features in dataset
1.2 To analyze and visualize timeline patterns of players by different groups and parameters
1.3 To create heat map based on the gaming zones
1.4 To visualize patterns based on Guilds they belong to

2.1.2 PERFORM PREDICTIVE ANALYTICS BY APPLYING MACHINE LEARNING MODELS

2.1 **Forecast the number of players expected in future time point** to plan resource capacity
2.2 **Predict player churning** to come up with steps to avoid future churn
2.3 **Recommend guilds** [groups to join] **to players** for effective gaming and to minimize churn

2.2 Data

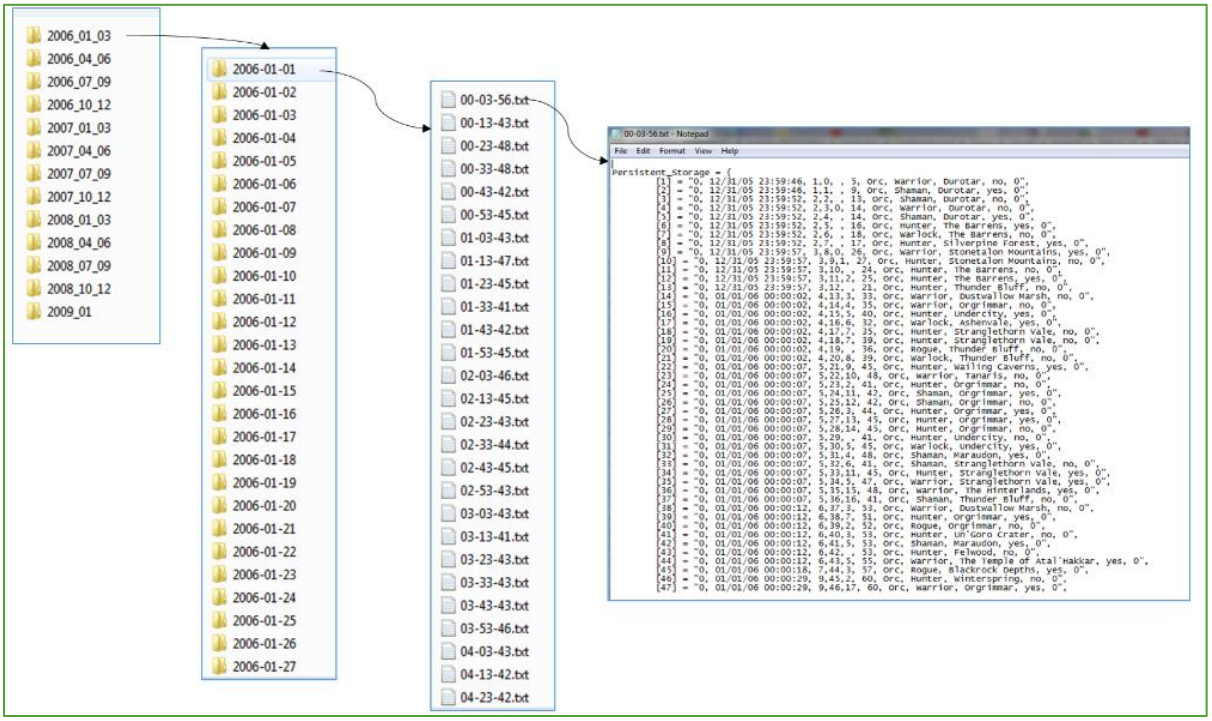
We have chosen an online game named “World of Warcraft” which we found to be most suitable for this Project. A large and scalable dataset with 3 years of player logs are released by Blizzard Entertainment for research purposes. We are using this dataset of our Project.

Data set Summary	
Attribute	**Value**
Data duration (in days)	1107
Sampling Rate per day	124
No. of Samples	138084
No. of Records (rows)	36,513,647
No. of Values (Data points)	438,163,764
Size of data (in GB)	3.4
Dataset Type	Logs
Format	Text Files
No. of Folders	1095

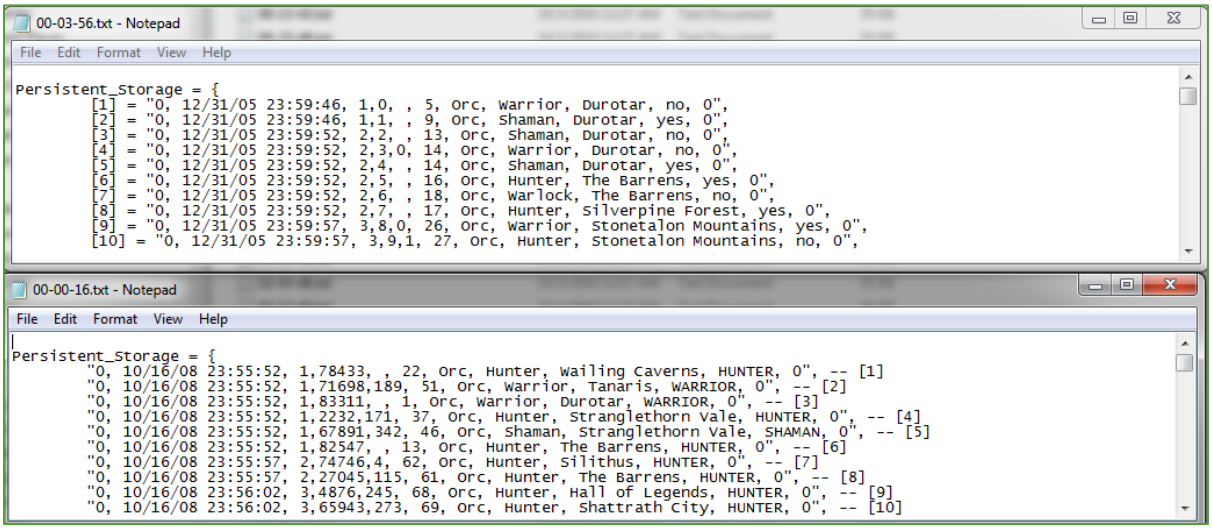
Field Description		
Field	**Description**	**Data Type**
Query Time	Date and time when logs were generated	integer
Query Seq. #	Sequence of queries	integer
Avatar ID	Unique id for each user	integer
Guild	Group id of the player	integer
Level	Game level of the player	integer
Race	Blood Elf, Orc, Tauren, Troll, Undead	String
Class	Death Knight, Druid, Hunter, Mage, Paladin, Priest, Rogue, Shaman, Warlock, Warrior	String
Zone	One of the 229 Zones in World of WarCraft game	String

2.3 Findings and Implications from Dataset

1. Dataset is in the following folder and file structure as shown in the below image



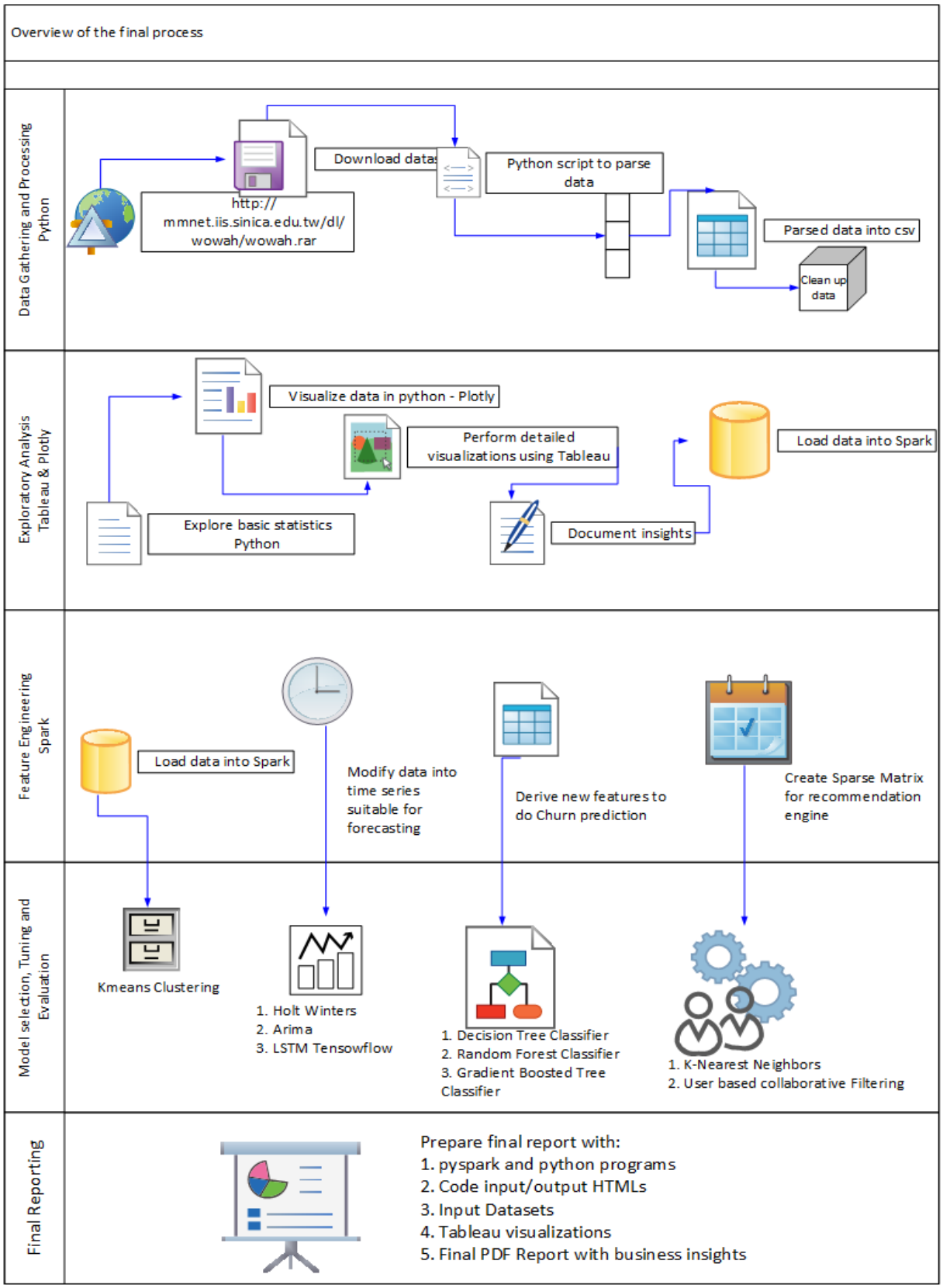
2. There are two different types of representations of logs within the folders



3. Both the types of logs need to be parsed and collected into a single csv file after removing irrelevant information.
4. Fortunately there are no data losses within the 3 year period of 2006 to 2009

III. OVERVIEW OF THE FINAL PROCESS

Described below is the detailed process flow with algorithms and techniques used :



IV. STEP-BY-STEP WALK THROUGH OF THE SOLUTION

Step 1: Parse Wow Logs

```
class wow_parser:
    def parse_logs(self,root_dir,output_file):
        import numpy as np
        import os
        import re
        strings = []
        for root, subdirs, files in os.walk(root_dir):
            for filename in files:
                if filename.endswith(".txt"):
                    file_path = os.path.join(root, filename)
                    with open(file_path) as f:
                        for line in f:
                            if "/" in line:
                                strings.extend(re.findall(r'"(.*?)"', line, re.DOTALL))
        thefile = open(output_file, 'w')
        for item in strings:
            thefile.write("%s\n" % item)
```

```
parse = wow_parser()
```

```
dirpath = "H:\WoWAH"
outputpath = "H:\Output\wowlogs.csv"
parse.parse_logs(root_dir = dirpath, output_file = outputpath)
```

	QueryTime	QuerySeq	AvatarID	Guild	Level	Race	Class	Zone
0	12/31/05 23:59:46	1	0		5	Orc	Warrior	Durotar
1	12/31/05 23:59:46	1	1		9	Orc	Shaman	Durotar
2	12/31/05 23:59:52	2	2		13	Orc	Shaman	Durotar
3	12/31/05 23:59:52	2	3	0	14	Orc	Warrior	Durotar
4	12/31/05 23:59:52	2	4		14	Orc	Shaman	Durotar

Step 2: Perform Exploratory Analysis and Clean up data

Following values are incorrect Warcraft races:
'373族', '547人', '3033','27410', '74622妖'
Let us look at the records which have these incorrect races.

```
df_incorrect_race = df[df['Race'].isin(['373族', '547人', '3033','27410', '74622妖'])]
```

```
df_incorrect_race.AvatarID.unique()
```

```
array([  373,   547,  3033, 27410, 74622], dtype=int64)
```

```
df_incorrect_race.count()
```

```
QueryTime    50085
QuerySeq     50085
AvatarID     50085
Guild        50085
Level        50085
Race         50085
Class        50085
Zone         50085
dtype: int64
```

Following values are incorrect Warcraft classes:
'482', '2400', '3485伊'
Let us look at the records which have these incorrect classes.

```
df_incorrect_class = df[df['Class'].isin(['482', '2400', '3485伊'])]
```

```
df_incorrect_class.AvatarID.unique()
```

```
array([ 482, 2400, 3485], dtype=int64)
```

```
df_incorrect_class.count()
```

```
QueryTime    376
QuerySeq     376
AvatarID     376
Guild        376
Level        376
Race         376
Class        376
Zone         376
dtype: int64
```

```
df_incorrect_zone.Zone.unique()
```

```
array([' 未知', ' 監獄', ' 時光洞穴', ' 達納蘇斯', '8585', '1608峽谷', '2029', '15641',
       '1007城', ' 北方海岸', ' 毒牙沼澤', ' 麥克那爾', '61477', ' 龍骨荒野', '1231崔茲',
       ' Dalaran競技場'], dtype=object)
```

```
df_incorrect_zone.count()
```

```
QueryTime    370298
QuerySeq     370298
AvatarID     370298
Guild        370298
Level        370298
Race         370298
Class        370298
Zone         370298
dtype: int64
```

```
df_incorrect_zone.AvatarID.nunique()
```

```
5441
```

```
(removed_records/total_records)*100
```

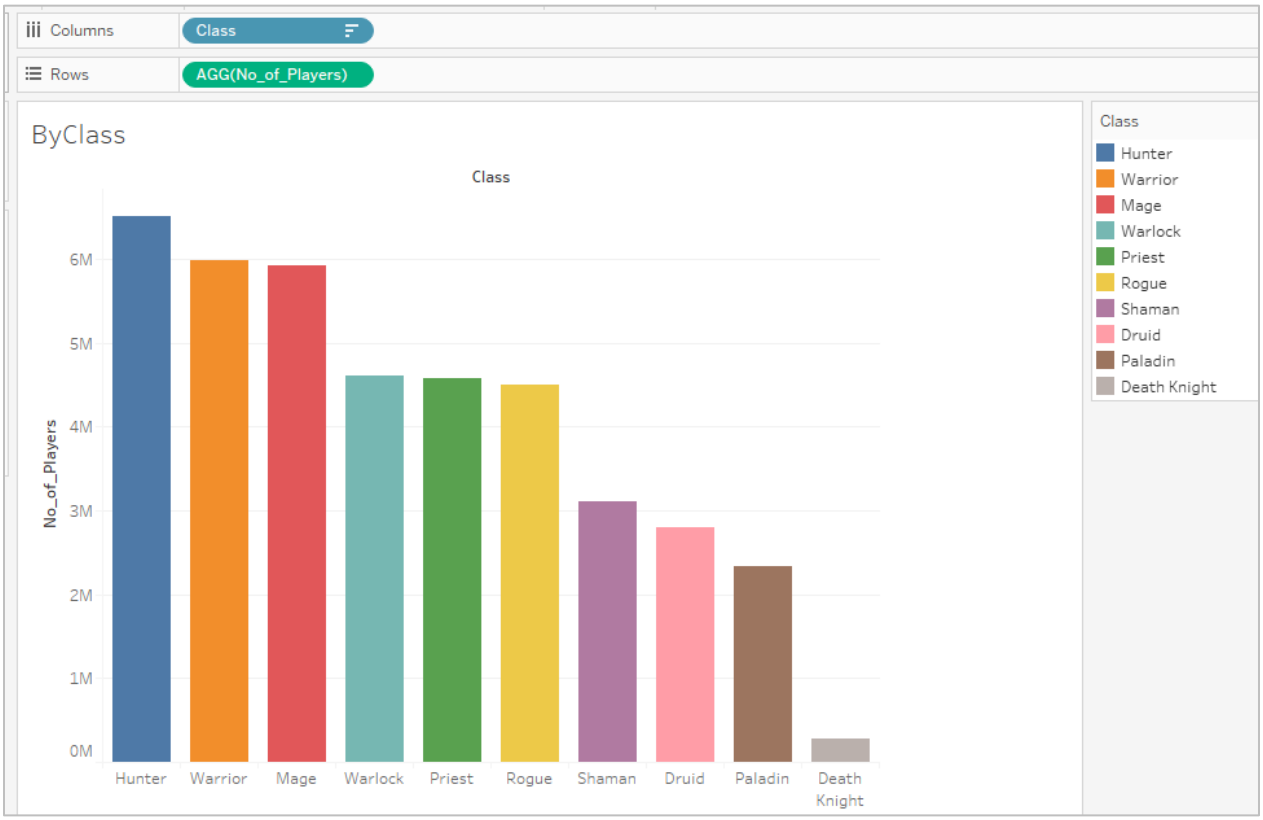
```
QueryTime    1.024008
QuerySeq     1.024008
AvatarID     1.024008
Guild        1.024008
Level        1.024008
Race         1.024008
Class        1.024008
Zone         1.024008
dtype: float64
```

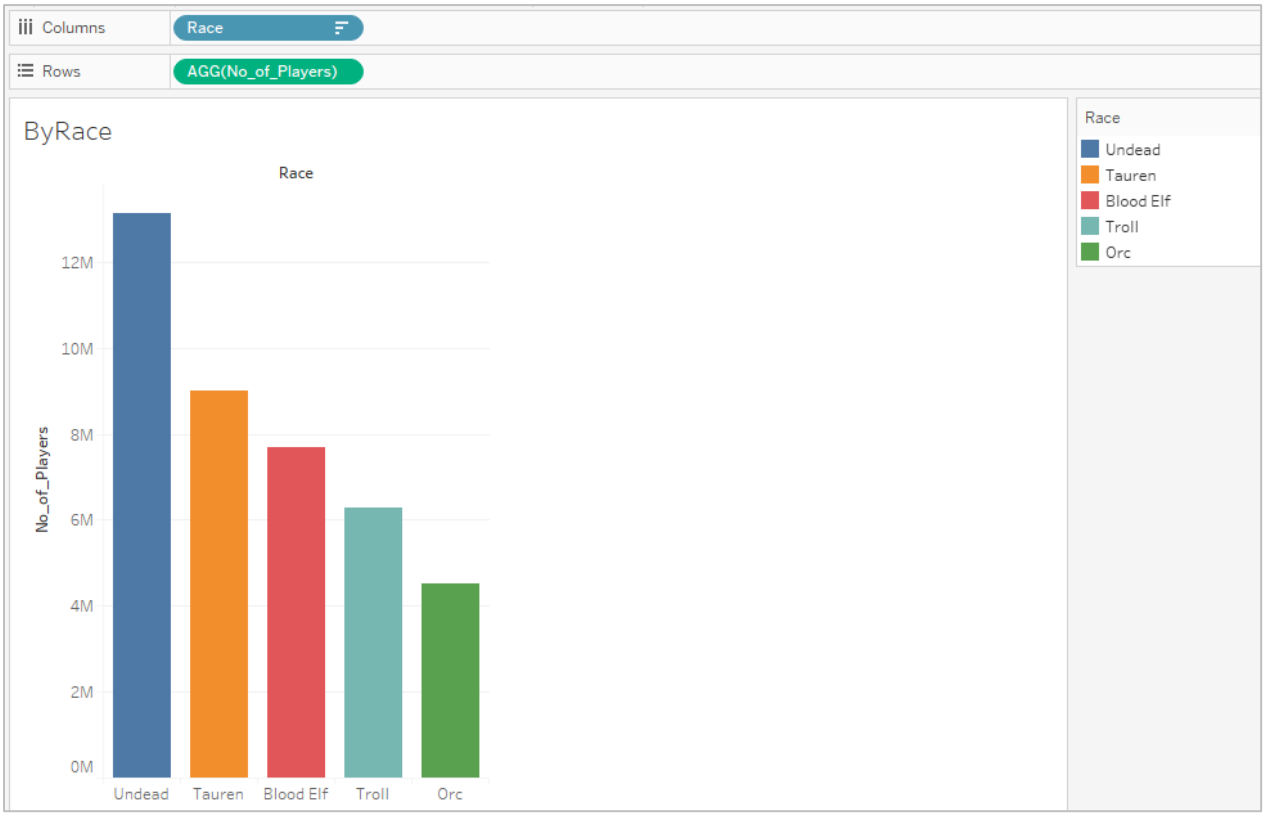
We will have to remove 1% of the records (420491 out of 41063255) to avoid incorrect analysis and inferences from warcraft logs. This data is relatively less compared to the total size of warcraft logs we have gathered.

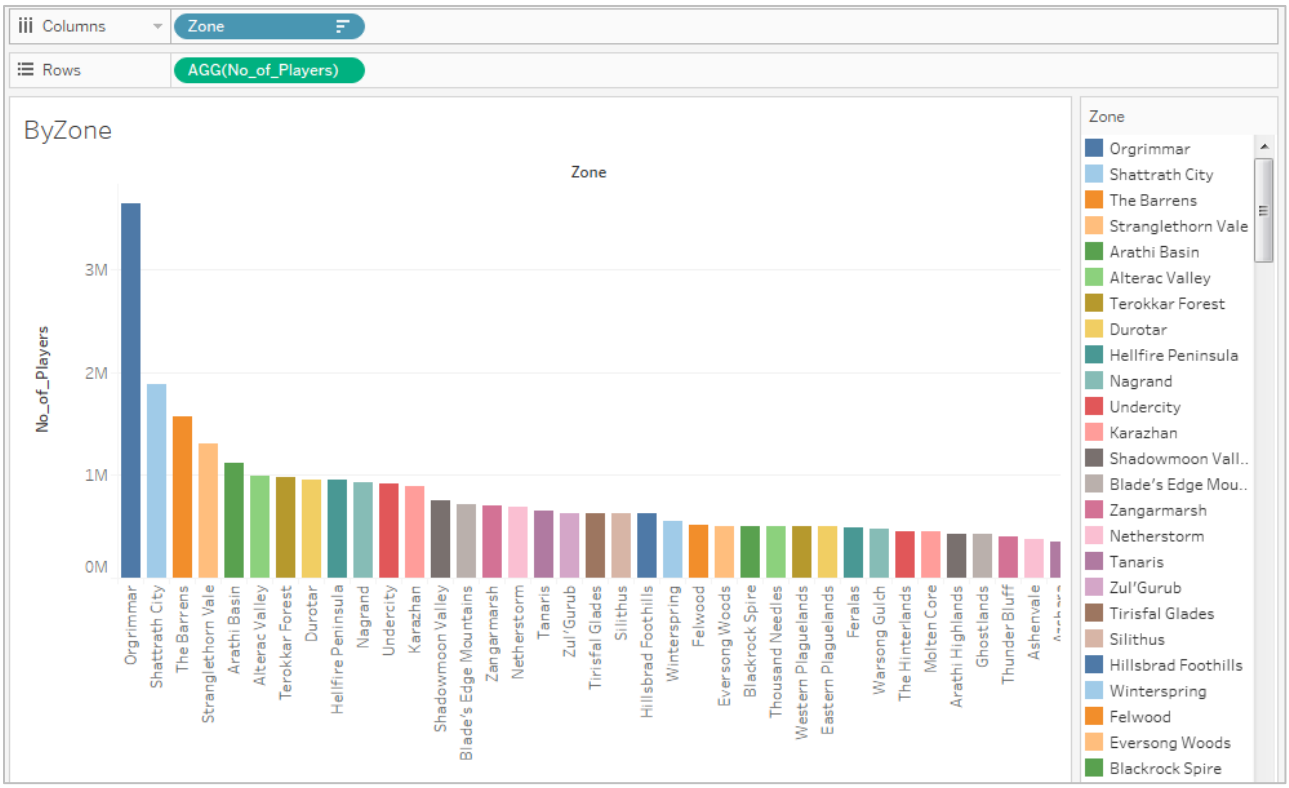
Save the final set of records into a new csv file to be used in further steps.

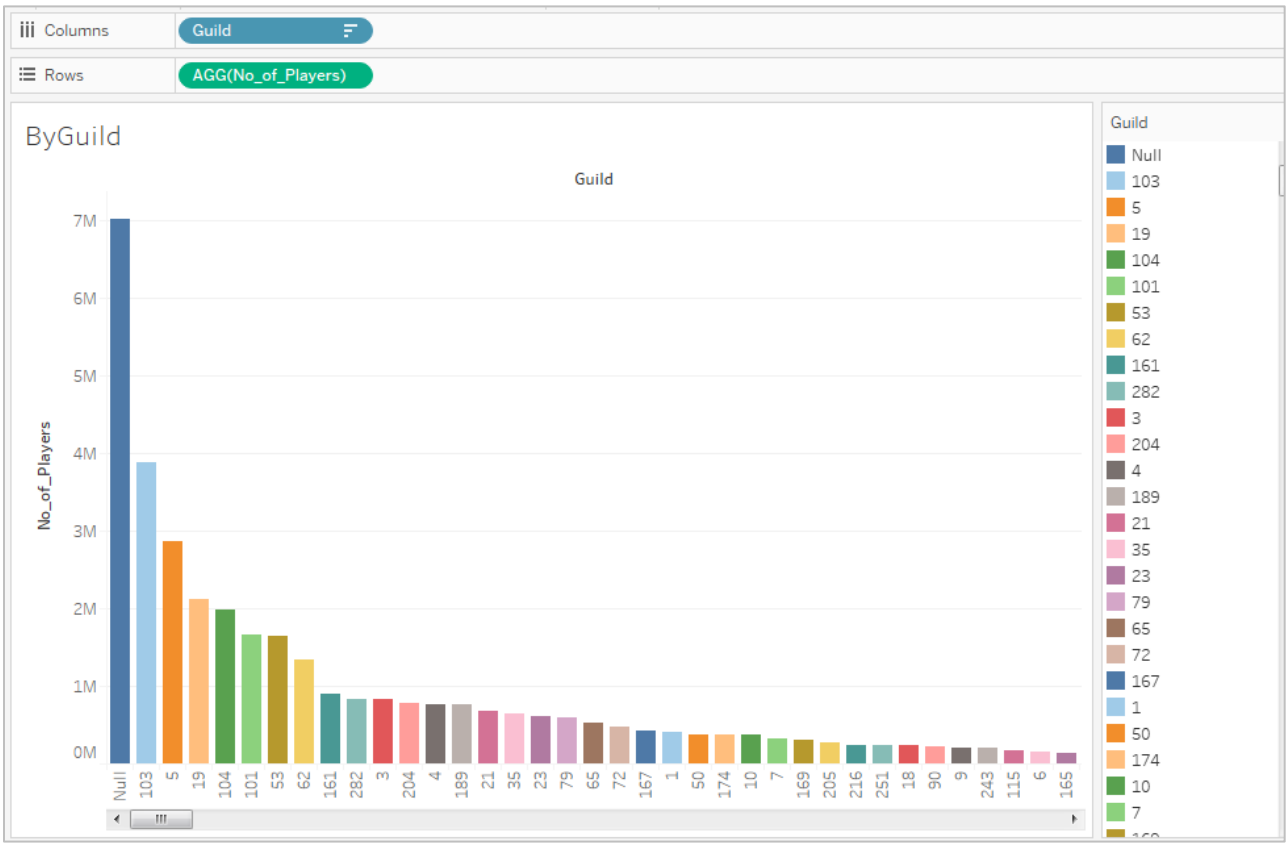
	QueryTime	QuerySeq	AvatarID	Guild	Level	Race	Class	Zone
40692952	01/10/09 05:08:48	56	36893	104	80	Blood Elf	Mage	Dalaran
40692953	01/10/09 05:08:48	56	39532	204	80	Blood Elf	Mage	The Storm Peaks
40692954	01/10/09 05:08:59	58	90033	502	80	Blood Elf	Death Knight	Sholazar Basin
40692955	01/10/09 05:08:59	58	87974	251	80	Blood Elf	Death Knight	Blade's Edge Mountains
40692956	01/10/09 05:08:59	58	86679	459	80	Blood Elf	Death Knight	Shadowmoon Valley

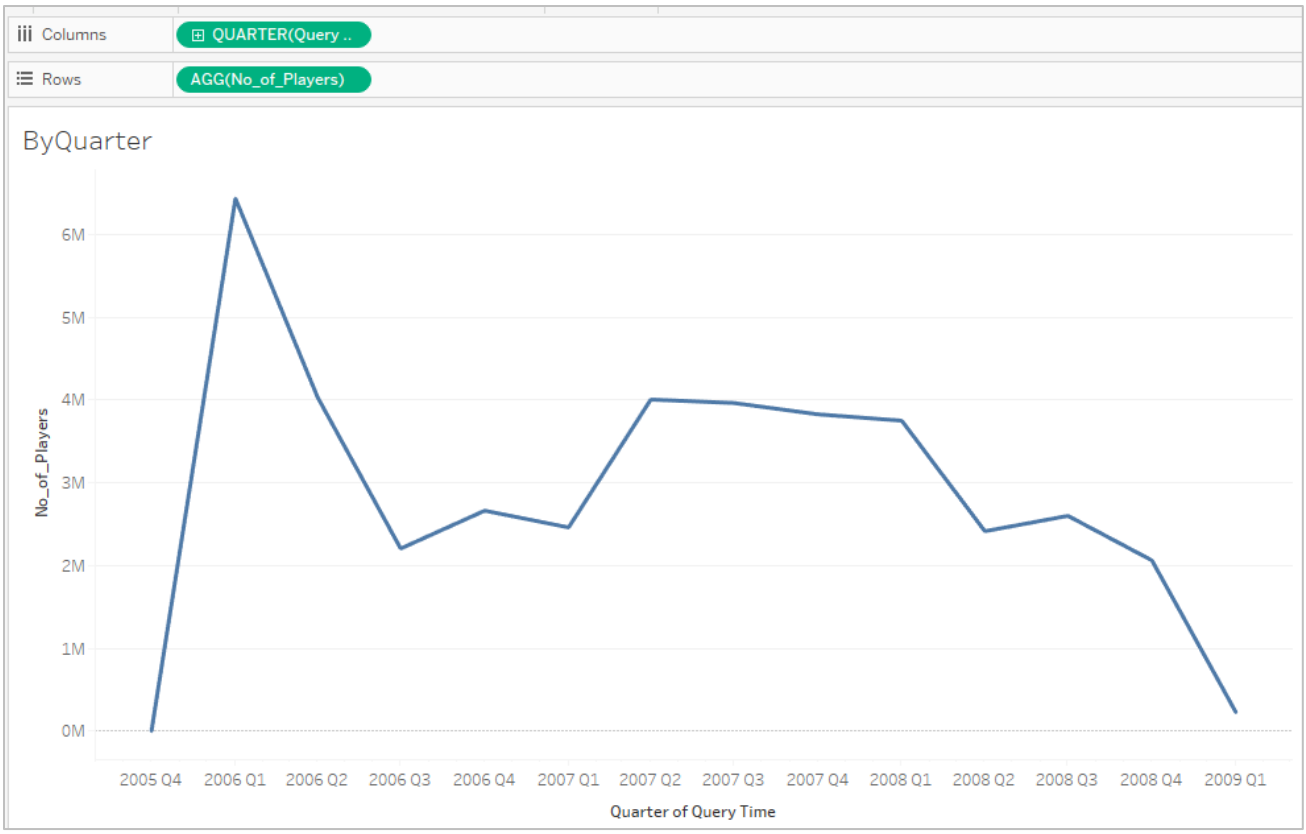
Step 3: Perform Exploratory Visualization - Tableau



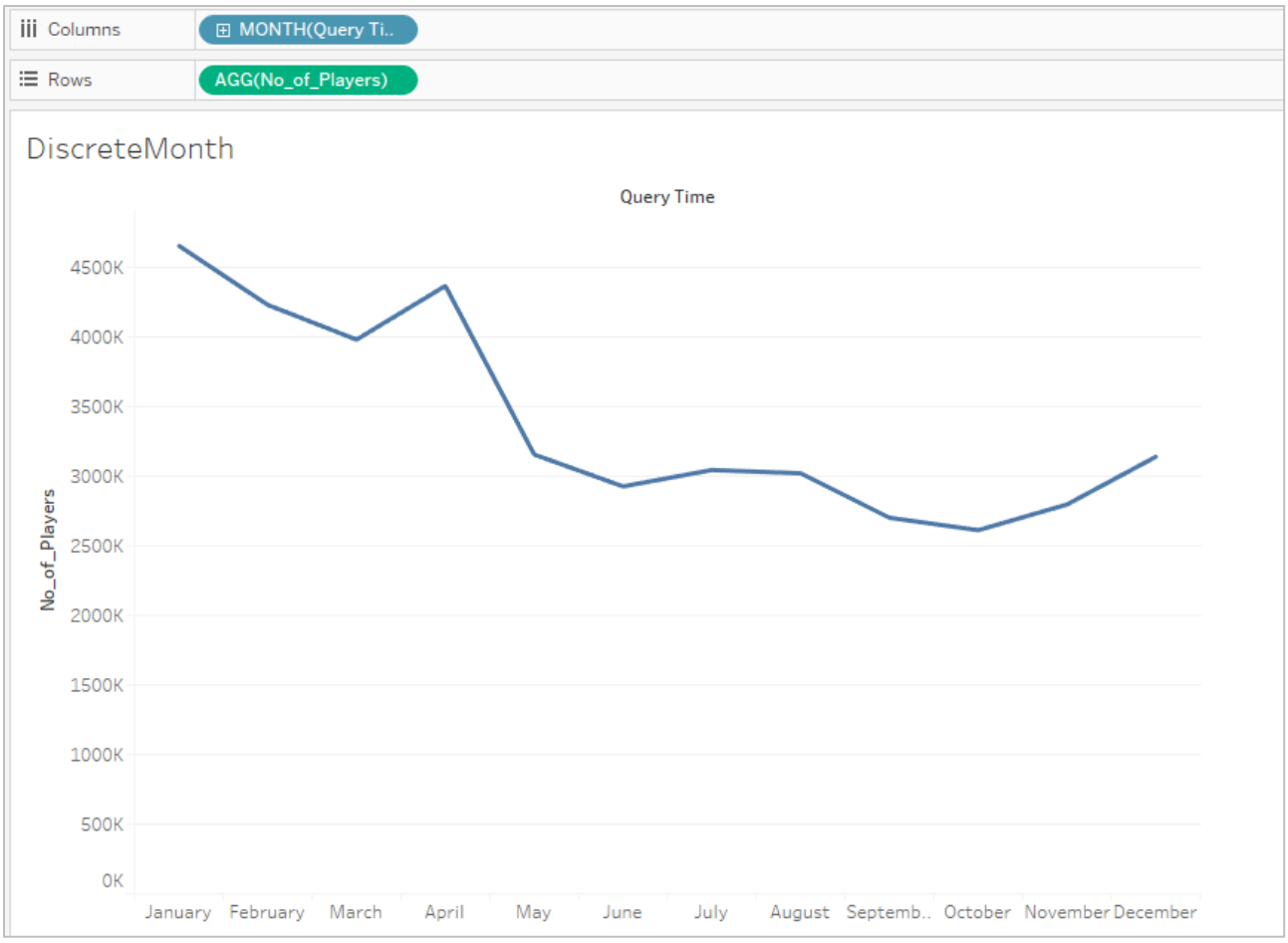


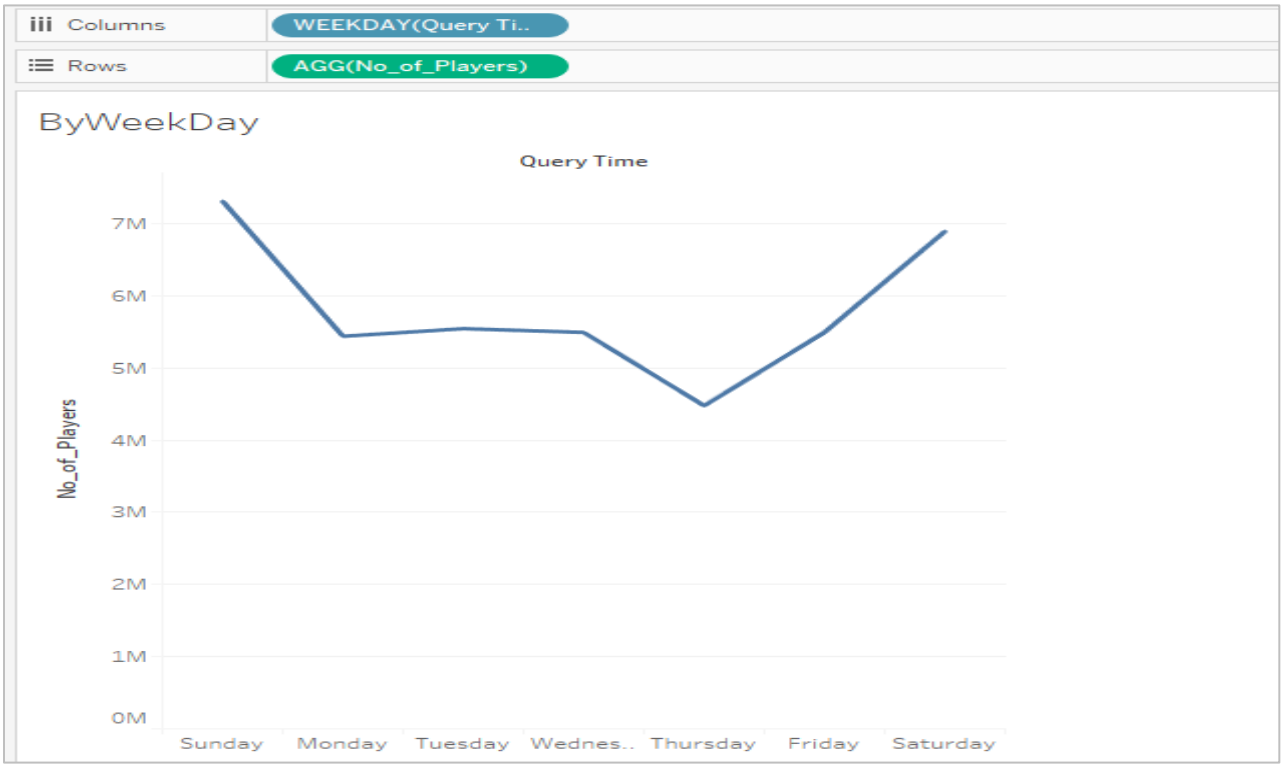

Columns
Zone
Rows
AGG(No_of_Players)
ByZone
Zone
No_of_Players
3M
2M
1M
0M
Orgrimmar
Shattrath City
The Barrens
Stranglethorn Vale
Arathi Basin
Alterac Valley
Terokkar Forest
Durotar
Hellfire Peninsula
Nagrand
Undercity
Karazhan
Shadowmoon Valley
Blade's Edge Mountains
Zangarmarsh
Netherstorm
Tanaris
Zul'Gurub
Tirisfal Glades
Silithus
Hillsbrad Foothills
Winterspring
Felwood
Eversong Woods
Blackrock Spire
Thousand Needles
Western Plaguelands
Eastern Plaguelands
Feralas
Warsong Gulch
The Hinterlands
Molten Core
Arathi Highlands
Ghostlands
Thunder Bluff
Ashenvale
Zone
Orgrimmar
Shattrath City
The Barrens
Stranglethorn Vale
Arathi Basin
Alterac Valley
Terokkar Forest
Durotar
Hellfire Peninsula
Nagrand
Undercity
Karazhan
Shadowmoon Vall..
Blade's Edge Mou..
Zangarmarsh
Netherstorm
Tanaris
Zul'Gurub
Tirisfal Glades
Silithus
Hillsbrad Foothills
Winterspring
Felwood
Eversong Woods
Blackrock Spire

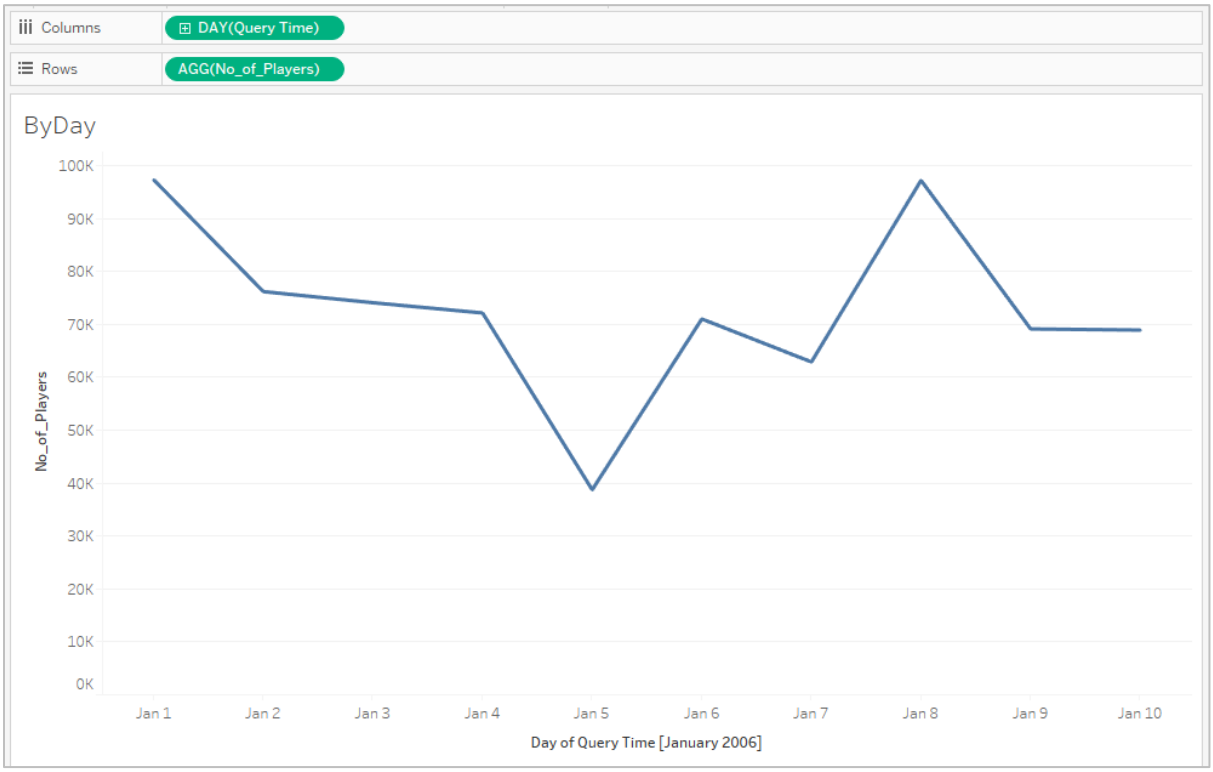

Columns
Guild
Rows
AGG(No_of_Players)
ByGuild
Guild
No_of_Players
7M
6M
5M
4M
3M
2M
1M
0M
Null
103
5
19
104
101
53
62
161
282
3
204
4
189
21
35
23
79
65
72
167
1
50
174
10
7
169
205
216
251
18
90
9
243
115
6
165
Guild
Null
103
5
19
104
101
53
62
161
282
3
204
4
189
21
35
23
79
65
72
167
1
50
174
10
7

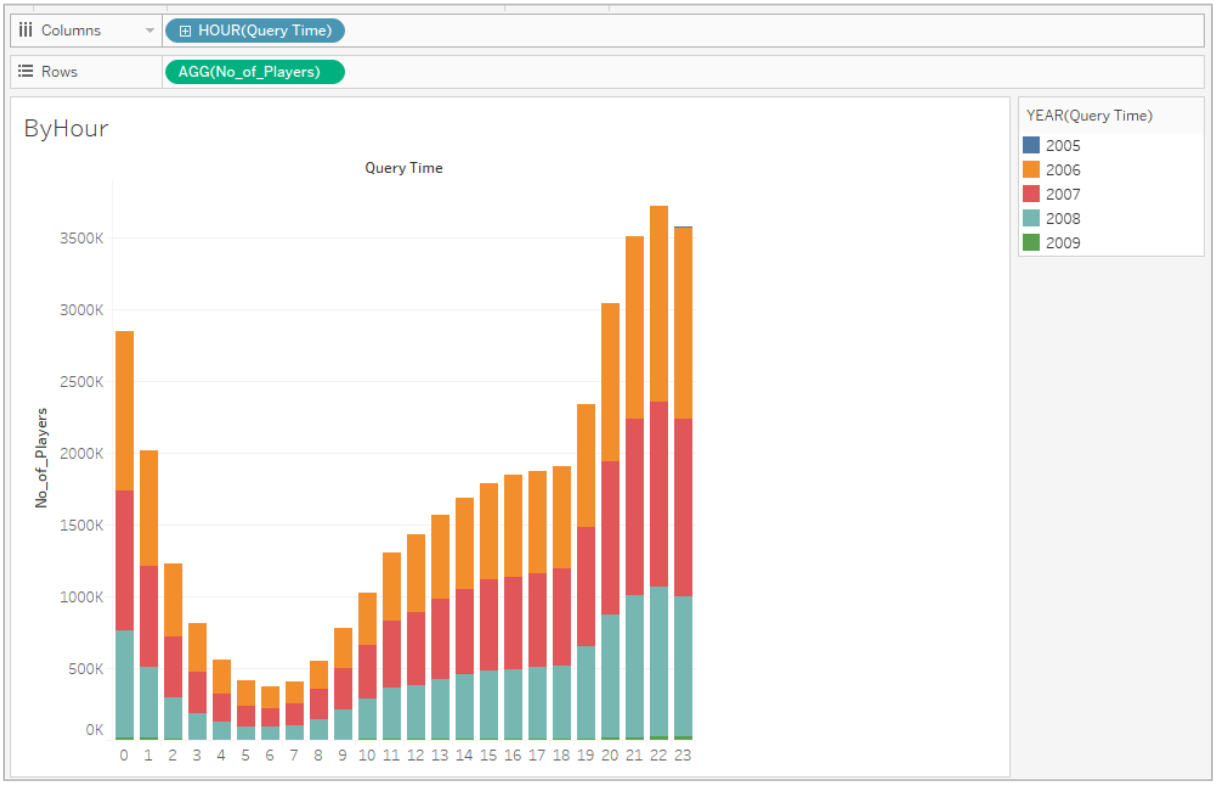

Columns
QUARTER(Query ..
Rows
AGG(No_of_Players)
ByQuarter
6M
5M
4M
3M
2M
1M
0M
No_of_Players
2005 Q4
2006 Q1
2006 Q2
2006 Q3
2006 Q4
2007 Q1
2007 Q2
2007 Q3
2007 Q4
2008 Q1
2008 Q2
2008 Q3
2008 Q4
2009 Q1
Quarter of Query Time


Columns
MONTH(Query Ti..
Rows
AGG(No_of_Players)
ByMonth
2000K
1500K
1000K
500K
0K
No_of_Players
November 2005
May 2006
November 2006
May 2007
November 2007
May 2008
November 2008
Month of Query Time


Columns
MONTH(Query Ti..
Rows
AGG(No_of_Players)
DiscreteMonth
Query Time
No_of_Players
4500K
4000K
3500K
3000K
2500K
2000K
1500K
1000K
500K
0K
January
February
March
April
May
June
July
August
Septemb..
October
November
December


Columns
WEEKDAY(Query Ti..
Rows
AGG(No_of_Players)
ByWeekDay
Query Time
No_of_Players
7M
6M
5M
4M
3M
2M
1M
0M
Sunday
Monday
Tuesday
Wednes..
Thursday
Friday
Saturday


Columns
DAY(Query Time)
Rows
AGG(No_of_Players)
ByDay
100K
90K
80K
70K
60K
50K
40K
30K
20K
10K
0K
No_of_Players
Jan 1
Jan 2
Jan 3
Jan 4
Jan 5
Jan 6
Jan 7
Jan 8
Jan 9
Jan 10
Day of Query Time [January 2006]


Columns
HOUR(Query Time)
Rows
AGG(No_of_Players)
ByHour
Query Time
YEAR(Query Time)
2005
2006
2007
2008
2009
3500K
3000K
2500K
2000K
1500K
1000K
500K
0K
No_of_Players
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

Step 4: Perform Feature Engineering (Python and Spark)

4.1 CREATE TIMESERIES INPUT FEATURES

Load cleaned up log files as a Spark dataframe.

```
newlogs_df = spark.read.csv(path = "/home/sulekhadileep/Documents/newlogs.csv", header = True,inferSchema = True)
```

Data is in minutes for each AvatarID. So, first count AvatarIDs by grouping them into minutes.

```
df2 = newlogs_df.groupBy(['QueryTime']).count()
```

Strip minutes and seconds from Query Time.

```
df2 = df2.withColumn('DateTime', df2['QueryTime'].substr(1,12))
```

```
df2
```

```
DataFrame[QueryTime: string, count: bigint, DateTime: string]
```

Now, take sum of players by grouping them by hours.

```
df3 = df2.groupBy(['DateTime']).agg({"count": "sum"})
```

```
df3
```

```
DataFrame[DateTime: string, sum(count): bigint]
```

```
df3 = df3.withColumn('Date', df2['DateTime'].substr(1,9))
```

```
df3 = df3.withColumnRenamed('sum(count)','PlayersCount')
```

Create a user defined function to get weekday for each Date since we are going to split data set by Week days and then do forecasting.

```
def get_weekday(date):
    import datetime
    import calendar
    month, day, year = (int(x) for x in date.split('/'))
    weekday = datetime.date(year, month, day)
    return calendar.day_name[weekday.weekday()]
```

```
spark.udf.register('get_weekday', get_weekday)
```

```
df3.createOrReplaceTempView("weekdays")
```

```
df4 = spark.sql("select DateTime, PlayersCount, get_weekday(Date) as Weekday from weekdays order by DateTime")
```

```
df4
```

```
DataFrame[DateTime: string, PlayersCount: bigint, Weekday: string]
```

```
df4.createOrReplaceTempView("completedata")
```

Collate all data for Sundays in a dataframe df_sunday

```
df_sunday = spark.sql("select DateTime, PlayersCount from completedata where Weekday = 'Sunday' order by DateTime")
```

Collate all data for Mondays in a dataframe df_monday

```
df_monday = spark.sql("select DateTime, PlayersCount from completedata where Weekday = 'Monday' order by DateTime")
```

Repeat the above two steps for all weekdays

```
df_tuesday = spark.sql("select DateTime, PlayersCount from completedata where Weekday = 'Tuesday' order by DateTime")
```

```
df_wednesday = spark.sql("select DateTime, PlayersCount from completedata where Weekday = 'Wednesday' order by DateTime")
```

```
df_thursday = spark.sql("select DateTime, PlayersCount from completedata where Weekday = 'Thursday' order by DateTime")
```

```
df_friday = spark.sql("select DateTime, PlayersCount from completedata where Weekday = 'Friday' order by DateTime")
```

```
df_saturday = spark.sql("select DateTime, PlayersCount from completedata where Weekday = 'Saturday' order by DateTime")
```

4.2 CREATE FEATURE SET 1 FOR CHURN PREDICTION – METHOD 1

```
from pyspark.sql.functions import unix_timestamp
wow_data = wow_data.withColumn('QuertTime_TS',unix_timestamp(wow_data['QueryTime'], "yy/dd/MM hh:mm:ss").cast("timestamp"))
wow_data = wow_data.withColumn('QuertTime_D',wow_data['QuertTime_TS'].cast("date"))
```

```
wow_data.registerTempTable( "wow_data" )
```

```
wow_data = sqlContext.sql( """select AvatarID,case when Guild = ' ' then 0 else 1 end as Guild,
                          case when Level between 1 and 23 then 2
                               when Level between 24 and 47 then 3 else 4 end as Level,Race,
                          case when Race = ' Orc' then 1 else 0 end as Race_Orc,
                          case when Race = ' Tauren' then 1 else 0 end as Race_Tauren,
                          case when Race = ' Troll' then 1 else 0 end as Race_Troll,
                          case when Race = ' Undead' then 1 else 0 end as Race_Undead,
                          case when Class = ' Warrior' then 1 else 0 end as Class_Warrior,
                          case when Class = ' Hunter' then 1 else 0 end as Class_Hunter,
                          case when Class = ' Rogue' then 1 else 0 end as Class_Rogue,
                          case when Class = ' Shaman' then 1 else 0 end as Class_Shaman,
                          case when Class = ' Warlock' then 1 else 0 end as Class_Warlock,
                          case when Class = ' Druid' then 1 else 0 end as Class_Druid,
                          case when Class = ' Mage' then 1 else 0 end as Class_Mage,
                          case when Class = ' Priest' then 1 else 0 end as Class_Priest,
                          QuertTime_D as LoginTime
from wow_data""" )
```

```
from pyspark.sql.functions import month, year, dayofmonth,concat
```

```
wow_data = wow_data.withColumn( 'LoginTime_i', (concat(dayofmonth(wow_data.LoginTime),
                                        month( wow_data.LoginTime ),year( wow_data.LoginTime ))).cast("integer"))
```

```
wow_data_Login = wow_data.select('AvatarID','LoginTime')
```

```
wow_data_Login.registerTempTable( "wow_data_Login" )
```

```
wow_data_Login =  sqlContext.sql( """select AvatarID,LoginTime,count(*) as Loginperday from wow_data_Login group by AvatarID,Log
inTime""")
```

```
wow_data_Login.count()
```

```
447127
```

```
wow_data.registerTempTable( "wow_data" )
```

```
wow_derived = sqlContext.sql( """select AvatarID,max(LoginTime) as Last_login_dt,min(LoginTime) as first_login_dt,datediff(max(L
oginTime),min(LoginTime)) as total_days_actv,count(distinct(LoginTime)) as actual_days_actv,
count(LoginTime)as Login_count from wow_data group by AvatarID""")
```

```
wow_derived.count()
```

```
26165
```

```
wow_derived.registerTempTable( "wow_derived" )
```

```
wow_data_Login.registerTempTable( "wow_data_Login" )
```

```
df_2 =sqlContext.sql("""select l.AvatarID,max(Total_days_actv) as Total_days_actv,max(Actual_days_actv) as Actual_days_actv,max
(Login_count) as Total_login,
                                    case when max(Level) = 2 then 1 else 0 end as Level_l ,
                                    case when max(Level) = 3 then 1 else 0 end as Level_m ,
                                    case when max(Level) = 4 then 1 else 0 end as Level_h ,
                                    max(Race_Orc) as Race_Orc,
                                    max(Race_Tauren) as Race_Tauren,
                                    max(Race_Troll) as Race_Troll,
                                    max(Race_Undead) as Race_Undead,
                                    max(Class_Warrior) as Class_Warrior,
                                    max(Class_Hunter) as Class_Hunter,
                                    max(Class_Rogue) as Class_Rogue,
                                    max(Class_Shaman) as Class_Shaman,
                                    max(Class_Warlock) as Class_Warlock,
                                    max(Class_Druid) as Class_Druid,
                                    max(Class_Mage) as Class_Mage,
                                    max(Class_Priest) as Class_Priest
                                    from wow_data l, wow_derived d where
                                    l.AvatarID = d.AvatarID
                                    group by l.AvatarID""")
```

New Features

```
+--------+--------------+---------------+-----------+-------+-------+-------+--------+-----------+----------+-----------+----
---------+-----------+----------+-----------+------------+-----------+----------+------------+
|AvatarID|Total_days_actv|Actual_days_actv|Total_login|Level_l|Level_m|Level_h|Race_Orc|Race_Tauren|Race_Troll|Race_Undead|Clas
s_Warrior|Class_Hunter|Class_Rogue|Class_Shaman|Class_Warlock|Class_Druid|Class_Mage|Class_Priest|
+--------+--------------+---------------+-----------+-------+-------+-------+--------+-----------+----------+-----------+----
---------+-----------+----------+-----------+------------+-----------+----------+------------+
|     148|           748|             66|       1202|      0|      0|      1|       0|          1|         0|          0|
         0|          0|         0|          1|           0|          0|         0|           0|
|     463|          3307|            158|       2412|      0|      0|      1|       0|          0|         0|          1|
         0|          0|         0|          0|           0|          0|         0|           1|
|     471|            12|             11|        253|      0|      0|      1|       1|          0|         0|          0|
         1|          0|         0|          0|           0|          0|         0|           0|
|     496|             0|              1|          2|      1|      0|      0|       1|          0|         0|          0|
         1|          0|         0|          0|           0|          0|         0|           0|
|     833|             0|              1|          1|      1|      0|      0|       1|          0|         0|          0|
         1|          0|         0|          0|           0|          0|         0|           0|
|    1088|          3297|             70|       1527|      0|      0|      1|       0|          1|         0|          0|
         1|          0|         0|          0|           0|          0|         0|           0|
|    1238|          2576|             93|       1018|      0|      0|      1|       0|          0|         0|          1|
         0|          0|         0|          0|           0|          0|         0|           1|
|    1342|          3676|             76|        658|      0|      1|      0|       0|          0|         1|          0|
         0|          0|         0|          1|           0|          0|         0|           0|
|    1580|          2581|            103|       1232|      0|      0|      1|       1|          0|         0|          0|
         0|          1|         0|          0|           0|          0|         0|           0|
|    1591|           741|             49|       1598|      0|      0|      1|       1|          0|         0|          0|
         0|          1|         0|          0|           0|          0|         0|           0|
```

4.3 CREATE FEATURE SET 2 FOR CHURN PREDICTION – METHOD 2

Create dummy variables and populate binary values for Race and Class

```
race_dummy = wow_data.select("Race").distinct().rdd.flatMap(lambda x: x).collect()
```

```
race_exprs = [when(col("Race") == category, 1).otherwise(0).alias(category)
         for category in race_dummy]
```

```
race_wow = wow_data.select("AvatarID", *race_exprs)
```

```
class_dummy = wow_data.select("Class").distinct().rdd.flatMap(lambda x: x).collect()
```

```
class_exprs = [when(col("Class") == category, 1).otherwise(0).alias(category)
         for category in class_dummy]
```

```
class_wow = wow_data.select("AvatarID", *class_exprs)
```

```
class_wow
```

```
+--------+--------+------+-------+-------------+--------+------+-----+-------+--------+-------+
|AvatarID| Warlock| Druid| Hunter| Death Knight| Paladin| Rogue| Mage| Priest| Warrior| Shaman|
+--------+--------+------+-------+-------------+--------+------+-----+-------+--------+-------+
|       0|       0|     0|      0|            0|       0|     0|    0|      0|       1|      0|
|       1|       0|     0|      0|            0|       0|     0|    0|      0|       0|      1|
|       2|       0|     0|      0|            0|       0|     0|    0|      0|       0|      1|
|       3|       0|     0|      0|            0|       0|     0|    0|      0|       1|      0|
|       4|       0|     0|      0|            0|       0|     0|    0|      0|       0|      1|
|       5|       0|     0|      1|            0|       0|     0|    0|      0|       0|      0|
|       6|       1|     0|      0|            0|       0|     0|    0|      0|       0|      0|
|       7|       0|     0|      1|            0|       0|     0|    0|      0|       0|      0|
|       8|       0|     0|      0|            0|       0|     0|    0|      0|       1|      0|
|       9|       0|     0|      1|            0|       0|     0|    0|      0|       0|      0|
|      10|       0|     0|      1|            0|       0|     0|    0|      0|       0|      0|
|      11|       0|     0|      1|            0|       0|     0|    0|      0|       0|      0|
|      12|       0|     0|      1|            0|       0|     0|    0|      0|       0|      0|
|      13|       0|     0|      0|            0|       0|     0|    0|      0|       1|      0|
```

Use df3 to calculate login dates

```
PlayerbyDate.createOrReplaceTempView("playerbydate")
```

```
PlayerbyDays.createOrReplaceTempView("playerbydays")
```

```
wow_data.createOrReplaceTempView("completedata")
```

```
dfx = spark.sql("select playerbydate.AvatarID, min(PlayerbyDate.Date) as FirstLoginDate, max(PlayerbyDate.Date) as LastLoginDate
 from playerbydate group by playerbydate.AvatarID")
```

```
timeformat = "yyyy-MM-dd HH:mm:ss"
tenure = unix_timestamp('LastLoginDate', format = timeformat) - unix_timestamp('FirstLoginDate', format = timeformat)
dfx = dfx.withColumn('PlayerTenure',tenure/(24*60*60))
```

```
dfx
```

```
+--------+-------------------+-------------------+------------+
|AvatarID|     FirstLoginDate|      LastLoginDate|PlayerTenure|
+--------+-------------------+-------------------+------------+
|   11033|2006-03-30 00:00:00|2006-09-06 00:00:00|       160.0|
|   18024|2006-07-19 00:00:00|2008-11-09 00:00:00|       844.0|
|    1342|2006-01-01 00:00:00|2009-01-07 00:00:00|      1102.0|
|   33412|2007-03-25 00:00:00|2008-04-02 00:00:00|       374.0|
|   30970|2007-02-11 00:00:00|2008-10-06 00:00:00|       603.0|
+--------+-------------------+-------------------+------------+
only showing top 5 rows
```

```
from pyspark.sql.functions import lit
lastlogcapture = '2009-01-10 00:00:00'
dfx = dfx.withColumn('LastLogCapture',lit(lastlogcapture))
```

```
nologin = unix_timestamp('LastLogCapture', format = timeformat) - unix_timestamp('LastLoginDate', format = timeformat)
dfx = dfx.withColumn('NotLoggedInFrom',nologin/(24*60*60))
```

```
dfx
```

```
+--------+-------------------+-------------------+------------+---------------+-------------------+
|AvatarID|     FirstLoginDate|      LastLoginDate|PlayerTenure|NotLoggedInFrom|     LastLogCapture|
+--------+-------------------+-------------------+------------+---------------+-------------------+
|   11033|2006-03-30 00:00:00|2006-09-06 00:00:00|       160.0|          857.0|2009-01-10 00:00:00|
|   18024|2006-07-19 00:00:00|2008-11-09 00:00:00|       844.0|           62.0|2009-01-10 00:00:00|
|    1342|2006-01-01 00:00:00|2009-01-07 00:00:00|      1102.0|            3.0|2009-01-10 00:00:00|
|   33412|2007-03-25 00:00:00|2008-04-02 00:00:00|       374.0|          283.0|2009-01-10 00:00:00|
|   30970|2007-02-11 00:00:00|2008-10-06 00:00:00|       603.0|           96.0|2009-01-10 00:00:00|
|    2866|2006-01-07 00:00:00|2009-01-09 00:00:00|      1098.0|            1.0|2009-01-10 00:00:00|
|   51393|2007-07-16 00:00:00|2008-10-25 00:00:00|       467.0|           77.0|2009-01-10 00:00:00|
|   46465|2007-06-04 00:00:00|2008-11-05 00:00:00|       520.0|           66.0|2009-01-10 00:00:00|
|   76110|2008-08-07 00:00:00|2008-08-28 00:00:00|        21.0|          135.0|2009-01-10 00:00:00|
|    8638|2006-02-28 00:00:00|2006-06-06 00:00:00|        98.0|          949.0|2009-01-10 00:00:00|
|    1645|2006-01-02 00:00:00|2008-11-10 00:00:00|      1043.0|           61.0|2009-01-10 00:00:00|
|   36131|2007-04-04 00:00:00|2009-01-06 00:00:00|       643.0|            4.0|2009-01-10 00:00:00|
|   22373|2006-11-03 00:00:00|2008-09-11 00:00:00|       678.0|          121.0|2009-01-10 00:00:00|
|   71527|2008-04-22 00:00:00|2008-05-12 00:00:00|        20.0|          243.0|2009-01-10 00:00:00|
|   34759|2007-04-03 00:00:00|2008-12-17 00:00:00|       624.0|           24.0|2009-01-10 00:00:00|
|   60769|2007-10-17 00:00:00|2009-01-02 00:00:00|       443.0|            8.0|2009-01-10 00:00:00|
|     463|2006-01-01 00:00:00|2008-03-16 00:00:00|       805.0|          300.0|2009-01-10 00:00:00|
|     148|2006-01-01 00:00:00|2008-12-10 00:00:00|      1074.0|           31.0|2009-01-10 00:00:00|
|   32414|2007-03-08 00:00:00|2009-01-10 00:00:00|       674.0|            0.0|2009-01-10 00:00:00|
|    8389|2006-02-25 00:00:00|2006-03-04 00:00:00|         7.0|         1043.0|2009-01-10 00:00:00|
+--------+-------------------+-------------------+------------+---------------+-------------------+
only showing top 20 rows
```

```
dfx.createOrReplaceTempView("datainprog")
```

```
dip = spark.sql("select datainprog.AvatarID, datainprog.PlayerTenure,case when datainprog.NotLoggedInFrom > 60  then 1 else 0 en
d as Churn \
                 from datainprog")
```

```
dip
```

```
+--------+------------+-----+
|AvatarID|PlayerTenure|Churn|
+--------+------------+-----+
|   11033|       160.0|    1|
|   18024|       844.0|    1|
|    1342|      1102.0|    0|
|   33412|       374.0|    1|
|   30970|       603.0|    1|
|    2866|      1098.0|    0|
|   51393|       467.0|    1|
|   46465|       520.0|    1|
|   76110|        21.0|    1|
|    8638|        98.0|    1|
|    1645|      1043.0|    1|
|   36131|       643.0|    0|
|   22373|       678.0|    1|
|   71527|        20.0|    1|
|   34759|       624.0|    0|
|   60769|       443.0|    0|
|     463|       805.0|    1|
|     148|      1074.0|    0|
|   32414|       674.0|    0|
|    8389|         7.0|    1|
+--------+------------+-----+
only showing top 20 rows
```

```
zone_wow = spark.sql("select max(AvatarID) as AvatarID, count(Zone) as ZonesPlayed\
                      from completedata group by AvatarID order by AvatarID")
```

```
zone_wow = zone_wow.drop_duplicates()
```

```
zone_wow
```

```
+--------+-----------+
|AvatarID|ZonesPlayed|
+--------+-----------+
|     260|        580|
|     443|      22488|
|     717|         34|
|    1495|        950|
|    1509|         61|
|    1902|          2|
|    1989|         12|
|    2011|        128|
|    2366|        401|
|    2533|          4|
|    2702|          2|
|    2985|        160|
|    3317|          2|
|    3577|          4|
|    3619|          4|
|    3764|         86|
|    4371|       5555|
|    5453|         64|
|    5541|        251|
|    5668|          2|
+--------+-----------+
only showing top 20 rows
```

New Features

```
+--------+-------------+-----------+-----------+----------------+-------------+-----------+----------+-----------+----------
---+-----------+
|AvatarID|Class_Warlock|Class_Druid|Class_Hunter|Class_DeathKnight|Class_Paladin|Class_Rogue|Class_Mage|Class_Priest|Class_Warr
ior|Class_Shaman|
+--------+-------------+-----------+-----------+----------------+-------------+-----------+----------+-----------+----------
---+-----------+
|      28|            0|          0|          1|               0|            0|          0|         0|          0|
  0|          0|
|     156|            0|          0|          0|               0|            0|          0|         0|          0|
  1|          0|
|     361|            0|          0|          0|               0|            0|          0|         0|          1|
  0|          0|
|     406|            0|          0|          0|               0|            0|          0|         0|          0|
  1|          0|
|     430|            0|          0|          1|               0|            0|          0|         0|          0|
  0|          0|
|     698|            0|          0|          0|               0|            0|          1|         0|          0|
  0|          0|
|     835|            0|          0|          0|               0|            0|          0|         0|          0|
  0|          1|
|    1177|            0|          0|          0|               0|            0|          0|         1|          0|
  0|          0|
|    1206|            1|          0|          0|               0|            0|          0|         0|          0|
  0|          0|
|    1222|            0|          0|          0|               0|            0|          0|         0|          0|
  1|          0|
|    1254|            1|          0|          0|               0|            0|          0|         0|          0|
  0|          0|
|    1609|            0|          1|          0|               0|            0|          0|         0|          0|
  0|          0|
|    2893|            0|          0|          0|               0|            0|          1|         0|          0|
  0|          0|
|    3085|            0|          0|          0|               0|            0|          1|         0|          0|
  0|          0|
|    3279|            0|          0|          1|               0|            0|          0|         0|          0|
```

```
FinalChurnDF
```

```
DataFrame[AvatarID: int, Class_Warlock: int, Class_Druid: int, Class_Hunter: int, Class_DeathKnight: int, Class_Paladin: int,
lass_Rogue: int, Class_Mage: int, Class_Priest: int, Class_Warrior: int, Class_Shaman: int, Race_Orc: int, Race_Tauren: int, R
ce_Undead: int, Race_BloodElf: int, Race_Troll: int, ZonesPlayed: bigint, LevelFlag: int, GuildFlag: int, DaysPlayed: bigint,
layerTenure: double, Churn: int]
```

4.4 CREATE FEATURE SET 1 FOR RECOMMENDATION ENGINE – METHOD 1

Loaded data needs to be seperately treated

Race & Class patterns for each Avatar ID are grouped and a new Column RaceClassCount is created. Likewise, Guild and Avatar ID are grouped to create a new column GuildCount

```
df_rc = df.groupby(['AvatarID','Race','Class']).size().reset_index()
#df_g = df.groupby(['AvatarID','Guild']).size().reset_index()
df = df.groupby(['AvatarID','Guild','Level','Race','Class']).size().reset_index()
```

```
df_rc.rename(columns = {0: 'RaceClassCount'},  inplace=True)
#df_g.rename(columns = {0: 'GuildCount'},  inplace=True)
df.rename(columns = {0: 'PatternCount'},  inplace=True)
```

Under all races same classes are there and we cannot count them seperately. So first, a Race Flag is created with unique number value and then a Class Flag with again unique values, such that when they are totalled they give unique values for Race and Class combinations.

```
def addraceflag(x):
    if x == ' Blood Elf':
        return 15
    elif x == ' Orc':
        return 3
    elif x == ' Tauren':
        return 6
    elif x == ' Troll':
        return 9
    elif x == ' Undead':
        return 12
```

```
df_rc['RaceFlag'] = df_rc.apply(lambda col: addraceflag(col['Race']), axis = 1)
```

```
def addclassflag(x):
    if x == ' Warrior':
        return 10
    elif x == ' Hunter':
        return 30
    elif x == ' Rogue':
        return 50
    elif x == ' Paladin':
        return 90
    elif x == ' Death Knight':
        return 100
    elif x == ' Shaman':
        return 20
    elif x == ' Warlock':
        return 40
    elif x == ' Druid':
        return 60
    elif x == ' Mage':
        return 70
    elif x == ' Priest':
        return 80
```

```
df_rc['ClassFlag'] = df_rc.apply(lambda col: addclassflag(col['Class']), axis = 1)
```

```
def addraceclass(row):
    if row['RaceFlag'] + row['ClassFlag'] == 25:
        return 'BloodElf & Warrior'
    elif row['RaceFlag'] + row['ClassFlag'] == 35:
        return 'Blood Elf & Shaman'
    elif row['RaceFlag'] + row['ClassFlag'] == 45:
        return 'Blood Elf & Hunter'
    elif row['RaceFlag'] + row['ClassFlag'] == 55:
        return 'Blood Elf & Warlock'
    elif row['RaceFlag'] + row['ClassFlag'] == 65:
        return 'Blood Elf & Rogue'
    elif row['RaceFlag'] + row['ClassFlag'] == 75:
        return 'Blood Elf & Druid'
    elif row['RaceFlag'] + row['ClassFlag'] == 85:
        return 'Blood Elf & Mage'
    elif row['RaceFlag'] + row['ClassFlag'] == 95:
        return 'Blood Elf & Priest'
    elif row['RaceFlag'] + row['ClassFlag'] == 105:
        return 'Blood Elf & Paladin'
    elif row['RaceFlag'] + row['ClassFlag'] == 115:
        return 'Blood Elf & Knight'
    elif row['RaceFlag'] + row['ClassFlag'] == 13:
        return 'Orc & Warrior'
    elif row['RaceFlag'] + row['ClassFlag'] == 23:
        return 'Orc & Shaman'
    elif row['RaceFlag'] + row['ClassFlag'] == 33:
        return 'Orc & Hunter'
    elif row['RaceFlag'] + row['ClassFlag'] == 43:
        return 'Orc & Warlock'
    elif row['RaceFlag'] + row['ClassFlag'] == 53:
        return 'Orc & Rogue'
    elif row['RaceFlag'] + row['ClassFlag'] == 63:
        return 'Orc & Druid'
```

Remove the users with no Guilds as we cannot recommend Guilds for them following the same process. For Users playing in Guilds already we will recommend new Guilds.

```
#df_nog = df_g['Guild'] == ' '
#df_g_final = df_g[~df_nog]

df_g = df['Guild'] >='0'
df_no_g = df[~df_g]
df_g = df[df_g]
```

```
df_no_g = df_no_g[['AvatarID','Level','Guild','PatternCount']]
df_g = df_g[['AvatarID','Level','Guild','PatternCount']]
```

```
if not df_no_g[df_no_g.duplicated(['AvatarID', 'Guild'])].empty:
    initial_rows = df_no_g.shape[0]
    print('Initial dataframe shape {0}'.format(df_no_g.shape))
    df_no_g = df_no_g.drop_duplicates(['AvatarID', 'Guild'])
    current_rows = df_no_g.shape[0]
    print('New dataframe shape {0}'.format(df_no_g.shape))
    print('Removed {0} rows'.format(initial_rows - current_rows))
```

```
Initial dataframe shape (32902, 4)
New dataframe shape (14953, 4)
Removed 17949 rows
```

```
if not df_g[df_g.duplicated(['AvatarID', 'Guild'])].empty:
    initial_rows = df_g.shape[0]
    print('Initial dataframe shape {0}'.format(df_g.shape))
    df_g = df_g.drop_duplicates(['AvatarID', 'Guild'])
    current_rows = df_g.shape[0]
    print('New dataframe shape {0}'.format(df_g.shape))
    print('Removed {0} rows'.format(initial_rows - current_rows))
```

4.5 CREATE FEATURE SET 2 FOR RECOMMENDATION – METHOD 2

Load cleaned up log files as a Spark dataframe.

```
newlogs_df = spark.read.csv(path = "newlogs.csv", header = True,inferSchema = True)
```

Data is in minutes for each AvatarID. So, first count AvatarIDs by grouping them into minutes.

```
df2 = newlogs_df.groupBy(['AvatarID','Guild','QueryTime']).count()
```

Strip time from Query Time.

```
df2 = df2.withColumn('Date', df2['QueryTime'].substr(1,9))
```

Now, take count of each player by grouping them by Avatar ID and Guild.

```
PlayerbyGuild = df2.groupBy(['AvatarID','Guild']).agg({"AvatarID": "count"})
```

```
PlayerbyGuild = PlayerbyGuild.withColumnRenamed('count(AvatarID)', 'GamesPlayed')
```

```
PlayerbyGuild.drop_duplicates()
```

```
122427
```

```
GuildData = PlayerbyGuild.groupby(['Guild']).agg({"GamesPlayed": "sum"})
```

```
GuildData = GuildData.withColumnRenamed('sum(GamesPlayed)','GuildGamesPlayed')
```

Convert Spark dataframe to Pandas.

```
import pandas as pd
```

```
PlayerinGuild = PlayerbyGuild.toPandas()
```

```
GuildGames = GuildData.toPandas()
```

```
matrix = PlayerinGuild.pivot(index = 'AvatarID', columns = 'Guild', values = 'GamesPlayed').fillna(0)
```

```
matrix.drop(u' ',axis=1, inplace=True)
```

Step 5: Perform Model Selection and Tuning (Python and Spark)

5.1 KMEANS CLUSTERING FOR DATA EXPLORATION

Creating Transformers

Encoding

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler, VectorIndexer
```

To encode the Guild, Level, Class, Race and Zone columns, we will use the OneHotEncoder method. However, the method cannot accept StringType columns; it can only deal with numeric types so first we will cast the columns to IntegerType.

```
categoricalColumns = ["Guild", "Race", "Class"]
stages = []
for i in categoricalColumns:
    # Category Indexing with StringIndexer
    stringIndexer = StringIndexer(inputCol=i, outputCol=i+"Index")
    # Use OneHotEncoder to convert categorical variables into binary SparseVectors
    encoder = OneHotEncoder(inputCol=i+"Index", outputCol=i+"classVec")
    stages += [stringIndexer, encoder]
```

Transform all features into a vector using VectorAssembler

```
assemblerinputsmap = map(lambda c: c + "classVec", categoricalColumns)
```

In Python 3 map cannot be added with list and hence using chain to convert it to list

```
from itertools import chain
classvecinputs = []
for i in chain(assemblerinputsmap):
    classvecinputs.append(i)
```

```
numericCols = ["Level"]
assemblerinputs = classvecinputs + numericCols
```

```
assembler = VectorAssembler(inputCols=assemblerinputs, outputCol="features")
stages += [assembler]
```

KMeans clustering

```
import pyspark.ml.clustering as clus
kmeans = clus.KMeans(k = 6,featuresCol='features')
stages += [kmeans]
```

```
# Create a Pipeline.
pipeline = Pipeline(stages=stages)
```

```
pipelineModel = pipeline.fit(cluster_df)
```

```
cluster_df = pipelineModel.transform(cluster_df)
```

```
cluster_df.printSchema()
```

```
root
 |-- Guild: string (nullable = true)
 |-- Level: integer (nullable = true)
 |-- Race: string (nullable = true)
 |-- Class: string (nullable = true)
 |-- AvatarID: integer (nullable = true)
 |-- Zone: string (nullable = true)
 |-- GuildIndex: double (nullable = true)
 |-- GuildclassVec: vector (nullable = true)
 |-- RaceIndex: double (nullable = true)
 |-- RaceclassVec: vector (nullable = true)
 |-- ClassIndex: double (nullable = true)
 |-- ClassclassVec: vector (nullable = true)
 |-- features: vector (nullable = true)
 |-- prediction: integer (nullable = true)
```

```
final_cluster_df = cluster_df["AvatarID", "Guild", "Level", "Race", "Class", "Zone","prediction"]
```

To store as spark partitioned files

final_cluster_df.repartition(1).write.csv('SixClusters.csv')

```
cluster = spark.read.csv(path = "SixClusters.csv", inferSchema = True)
```

```
cluster.printSchema()
```

```
root
 |-- _c0: integer (nullable = true)
 |-- _c1: integer (nullable = true)
 |-- _c2: integer (nullable = true)
 |-- _c3: string (nullable = true)
 |-- _c4: string (nullable = true)
 |-- _c5: string (nullable = true)
 |-- _c6: integer (nullable = true)
```

```
cluster = cluster.withColumnRenamed("_c0","AvatarID").withColumnRenamed("_c1","Guild").withColumnRenamed("_c2","Level").withColu
mnRenamed("_c3","Race").withColumnRenamed("_c4","Class").withColumnRenamed("_c5","Zone").withColumnRenamed("_c6","Prediction")
```

```
cluster.show(5)
```

```
+--------+-----+-----+----+-------+-------+----------+
|AvatarID|Guild|Level|Race|  Class|   Zone|Prediction|
+--------+-----+-----+----+-------+-------+----------+
|       0| null|    5| Orc|Warrior|Durotar|         3|
|       1| null|    9| Orc| Shaman|Durotar|         3|
|       2| null|   13| Orc| Shaman|Durotar|         3|
|       3|    0|   14| Orc|Warrior|Durotar|         3|
|       4| null|   14| Orc| Shaman|Durotar|         3|
+--------+-----+-----+----+-------+-------+----------+
only showing top 5 rows
```

Now we have the data grouped into 6 clusters. Further detailed Exploratory Analysis on these clusters are performed in the Tableau File: ClusterAnalysis.twbx

5.2 TIME SERIES FORECASTING OF HOURLY DATA USING HOLT WINTERS MODEL

Created a package to calculate Holt Winters Forecasting.
Install using the following command:
pip install **sulekha_holtwinters**

```
from sulekha_holtwinters import holtwinters as hw
```

```
hd = hw.holtwinters()
```

Set up the Pyspark environment required for running the model

```
#Pyspark setup
from pyspark import SparkContext
from pyspark.sql import SQLContext
from pyspark.sql import Row, SparkSession
from pyspark.sql.types import NumericType
```

```
sc = SparkContext.getOrCreate()
spark = SQLContext(sc)
```

Time series Forecasting model building

Sunday Forecast

```
df_sunday = spark.read.csv(path = "timeseries_wow_hourly_sunday.csv", header = True,inferSchema = True)
```

Try various values for the optimization parameters - alpha, beta and gamma

Trial 1:

```
Observed, Predictions, Level, Trend, Seasonality = hd.holtwinters_additive(df_sunday,0.05,0.06,0.07,24,24)
```

Calculate Mean Average Percentage Error between actual and forecasted values

```
hd.MAPE(Observed, Predictions)
```

```
36.568545344429694
```

Trail 2:

```
Observed, Predictions, Level, Trend, Seasonality = hd.holtwinters_additive(df_sunday,0.15,0.16,0.17,24,24)
```

```
hd.MAPE(Observed, Predictions)
```

```
20.34377185289498
```

Trial 3:

```
Observed, Predictions, Level, Trend, Seasonality = hd.holtwinters_additive(df_sunday,0.145,0.146,0.247,24,24)
```

```
hd.MAPE(Observed, Predictions)
```

```
18.875780664814123
```

Trial 4:

```
Observed, Predictions, Level, Trend, Seasonality = hd.holtwinters_additive(df_sunday,0.865,0.01,0.865,24,24)
```

```
hd.MAPE(Observed, Predictions)
```

```
0.7025811154520147
```

Use the below function to find out the most suitable combination of alpha, beta and gamma

```
Accuracy, Alpha, Beta, Gamma = hd.BestFit_Additive(df_sunday, interval=865, denominator=1000,L = 24,n_predictions = 24)
```

```
Accuracy
```

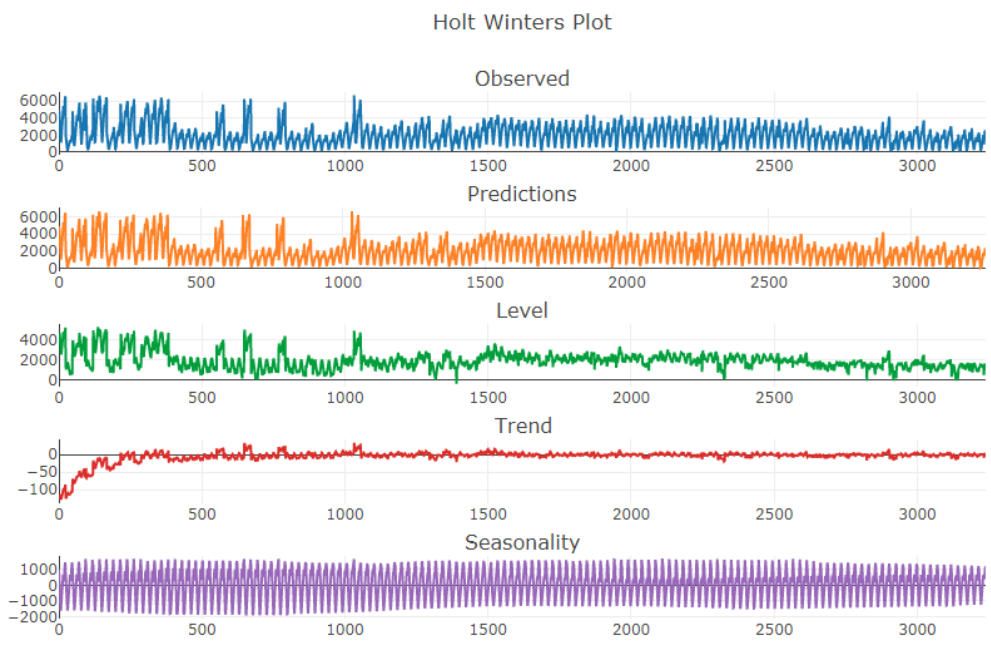
```
89.18893460158974
```

```
Alpha,Beta, Gamma
```

```
(0.865, 0.0, 0.865)
```

Create graphs to view the quality of Trend, Level, Sesonality, Predictions and Observations

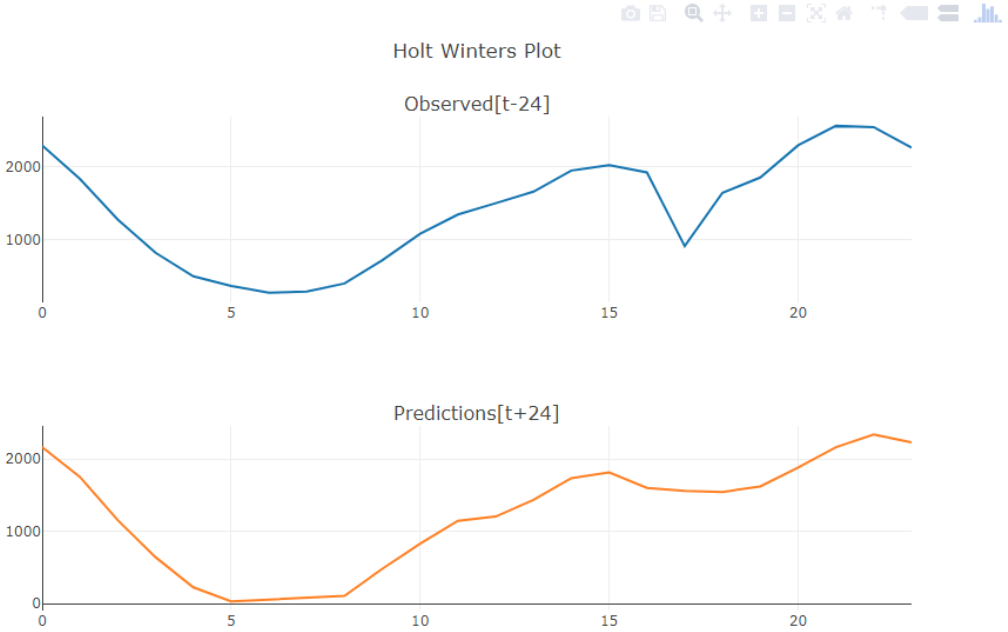
```
model1 = hd.holtwinters_additive(df_sunday,0.865,0.01,0.865,24,24)
hd.CreateGraphs(model1)
```



Plot last 24 predictions in comparison to last 24 observations to check the quality of prediction model

```
def plot(Observed,Predictions):
    from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
    from plotly import tools
    init_notebook_mode(connected=True) #Plotly offline
    from plotly import graph_objs as go
    from plotly import figure_factory
    #input
    Observed = Observed
    Predictions = Predictions
    observed = go.Scatter(y = Observed )
    predictions = go.Scatter(y = Predictions)
    data = [observed, predictions]
    fig = tools.make_subplots(rows=2, cols=1, subplot_titles = ("Observed[t-24]","Predictions[t+24]"),print_grid=False)
    fig.append_trace(observed, 1, 1)
    fig.append_trace(predictions, 2, 1)
    fig['layout'].update(height=600, width=900, title='Holt Winters Plot', showlegend=False)
    return iplot(fig)
```

```
plot(Observed[-24:], Predictions[-24:])
```



Verify RMSE of n periods predicted in future

To perform this check, data for one particular day (24 timepoints) is removed from the dataset and the remaining data is tested with Holt Winters forecasting and a future prediction period of n = 24 timepoints.

```
train = spark.read.csv(path = "timeseries_wow_hourly_train.csv", header = True,inferSchema = True)
test = spark.read.csv(path = "timeseries_wow_hourly_test.csv", header = True,inferSchema = True)
```

```
Observed, Predictions, Level, Trend, Seasonality = hd.holtwinters_additive(train,0.9,0.009,0.865,24,24)
```

```
observed = [p.PlayersCount for p in test.select("PlayersCount").collect()]
```

```
preds = Predictions[-24:]
```

Forecast RMSE

```
hd.Accuracy(Observed, Predictions)
```

```
99.34744290252247
```

```
hd.MAPE(Observed, Predictions)
```

```
0.6525570974775063
```

```
hd.RMSE(Observed,Predictions)
```

```
1087.2973261090815
```

Future Value RMSE

```
hd.RMSE(observed,preds)
```

```
2974.269556489125
```

```
hd.MAPE(observed,preds)
```

```
51.51014677728935
```

Insights:

This RMSE and MAPE values are much lesser compared to the values from ARIMA model and LSTM Tensorflow attempted.
Hence Holt WInters Triple Exponential Smoothing - Additive method seems to be more dependable for wow dataset timeseries forecasting.

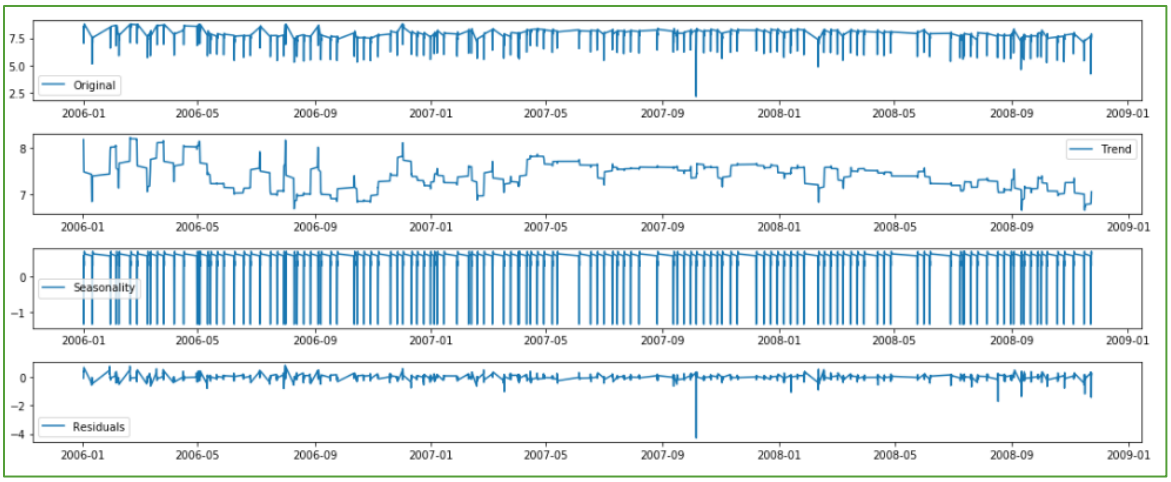
Accuracy of this model is highly dependant on the choice of values chosen for alpha, beta and gamma parameters. Eventhough there is a function available to calculate this value, choosing the optimal parameters is more based on trial and error.
Also, this model is more suitable for forecasting short term time points in future. If the model is attempted to predicted for longer term periods in future, accuracy would be very less and values might just show a linear trend since historical data for these kind of games would become stale as the time increases.

5.3 TIME SERIES FORECASTING OF HOURLY DATA USING ARIMA MODEL

```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(ts_log, freq = 24)

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.subplot(411)
plt.plot(ts_log, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```

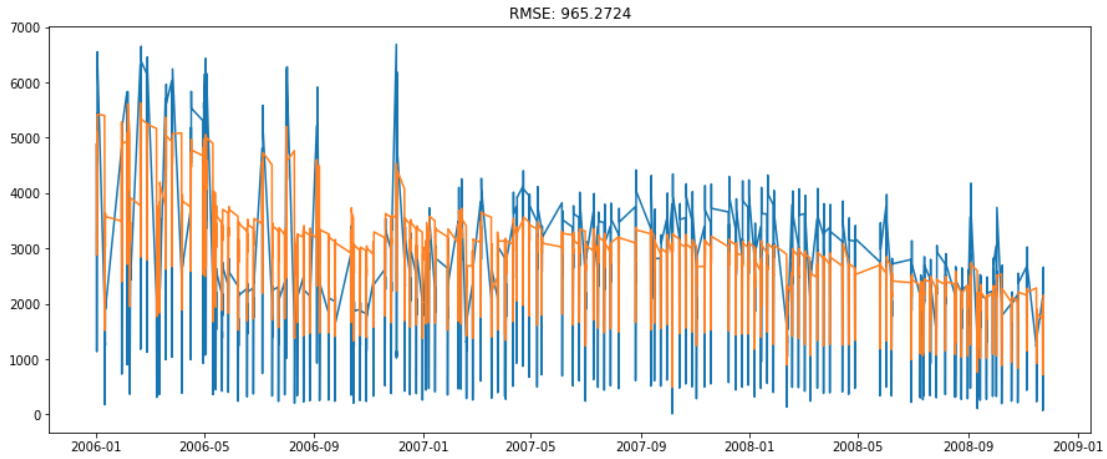



```
model = ARIMA(ts_log, order=(0, 1, 2))
results_MA = model.fit(disp=-1)
plt.plot(ts_log_diff)
plt.plot(results_MA.fittedvalues, color='red')
plt.title('RSS: %.4f'% sum((results_MA.fittedvalues-ts_log_diff)**2))
```

```
Text(0.5,1,'RSS: 250.6376')
```

```
predictions_ARIMA = np.exp(predictions_ARIMA_log)
plt.plot(ts)
plt.plot(predictions_ARIMA)
plt.title('RMSE: %.4f'% np.sqrt(sum((predictions_ARIMA-ts)**2)/len(ts)))
```

```
Text(0.5,1,'RMSE: 965.2724')
```




```
def MAPE(Observed, Predictions):
    n = len(Observed)
    MAPE = []
    for i in range(n):
        MAPE.append(abs((Observed[i] - Predictions[i]/Observed[i])))
    return sum(MAPE)*100/n
```

```
MAPE(ts,predictions_ARIMA)
```

```
205000.07605070341
```

5.4 TIME SERIES FORECASTING OF HOURLY DATA USING LSTM TENSOWFLOW

```
# transform data to be stationary
raw_values = series.values
diff_values = difference(raw_values, 1)
```

```
# transform data to be supervised learning
supervised = timeseries_to_supervised(diff_values, 1)
supervised_values = supervised.values
```

```
# split data into train and test-sets
train, test = supervised_values[0:-12], supervised_values[-12:]
```

```
# transform the scale of the data
scaler, train_scaled, test_scaled = scale(train, test)
```

```
# fit the model
lstm_model = fit_lstm(train_scaled, 1, 10, 4)
```

```
# forecast the entire training dataset to build up state for forecasting
train_reshaped = train_scaled[:, 0].reshape(len(train_scaled), 1, 1)
lstm_model.predict(train_reshaped, batch_size=1)
```

```
array([[ 0.06892812],
       [ 0.08625554],
       [ 0.08475064],
       ...,
       [-0.65559006],
       [-0.65664989],
       [-0.66087061]], dtype=float32)
```

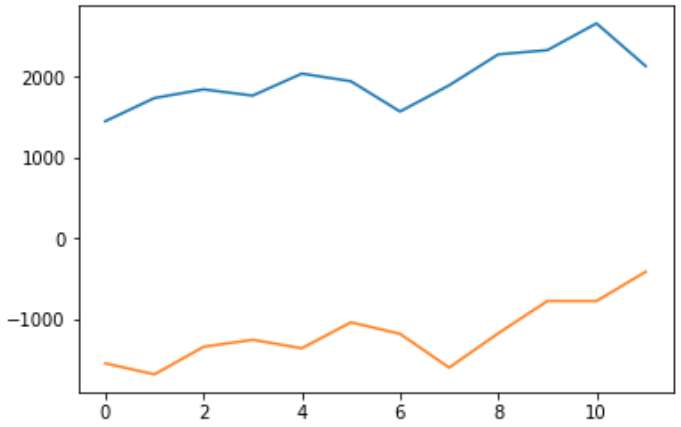
```
# walk-forward validation on the test data
predictions = list()
for i in range(len(test_scaled)):
    # make one-step forecast
    X, y = test_scaled[i, 0:-1], test_scaled[i, -1]
    yhat = forecast_lstm(lstm_model, 1, X)
    # invert scaling
    yhat = invert_scale(scaler, X, yhat)
    # invert differencing
    yhat = inverse_difference(raw_values, yhat, len(test_scaled)+1-i)
    # store forecast
    predictions.append(yhat)
    expected = raw_values[len(train) + i + 1]
    print('Time=%d, Predicted=%f, Expected=%f' % (i+1, yhat, expected))
```

```
Time=1, Predicted=-1544.160160, Expected=1450.000000
Time=2, Predicted=-1680.239737, Expected=1737.000000
Time=3, Predicted=-1339.753761, Expected=1844.000000
Time=4, Predicted=-1252.624352, Expected=1767.000000
Time=5, Predicted=-1357.669659, Expected=2039.000000
Time=6, Predicted=-1037.223542, Expected=1944.000000
Time=7, Predicted=-1179.360018, Expected=1570.000000
Time=8, Predicted=-1596.600141, Expected=1894.000000
Time=9, Predicted=-1172.181445, Expected=2277.000000
Time=10, Predicted=-773.033244, Expected=2329.000000
Time=11, Predicted=-773.771358, Expected=2660.000000
Time=12, Predicted=-411.195297, Expected=2132.000000
```

```
#report performance
rmse = sqrt(mean_squared_error(raw_values[-12:], predictions))
print('Test RMSE: %.3f' % rmse)
```

```
Test RMSE: 3160.213
```

```
# line plot of observed vs predicted
pyplot.plot(raw_values[-12:])
pyplot.plot(predictions)
pyplot.show()
```



Model accuracy is less compared to HoltWInters method.

5.5 CHURN PREDICTION – METHOD 1 ON FEATURE SET 1

Applying the Decision Tree Algorithm ¶

The prepared data set is split in to training and testing sets. A decision tree classifier model is then generated using the training data, using a maxDepth of 2, to build a "shallow" tree. The tree depth can be regarded as an indicator of model complexity.

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.tree import DecisionTree

def labelData(data):
    # label: row[end], features: row[0:end-1]
    return data.rdd.map(lambda row: LabeledPoint(row[-1], row[:-1]))

training_data, testing_data = labelData(wow_data_dt).randomSplit([0.8, 0.2])

model = DecisionTree.trainClassifier(training_data, numClasses=2, maxDepth=2,
                                     categoricalFeaturesInfo = {0:2,1:2,2:2,3:2,4:2,5:2,6:2,7:2,8:2,9:2,10:2,11:2,12:2,13:2,14:2
},
                                     impurity='gini', maxBins=32)

print model.toDebugString()
```

```
DecisionTreeModel classifier of depth 2 with 7 nodes
  If (feature 0 in {0.0})
   If (feature 1 in {1.0})
    Predict: 1.0
   Else (feature 1 not in {1.0})
    Predict: 1.0
  Else (feature 0 not in {0.0})
   If (feature 14 in {1.0})
    Predict: 1.0
   Else (feature 14 not in {1.0})
    Predict: 1.0
```

The toDebugString() function provides a print of the trees decision nodes and final prediction outcomes at the end leafs. We can see that features 14 and 1 are used for decision making and should thus be considered as having high predictive power to determine a user's likeliness to churn.The featues map to the perticular Race and whether the user has reached a medium level in play. Decision trees are often used for feature selection because they provide an automated mechanism for determining the most important features (those closest to the tree root).

```
print 'Feature 14:', wow_data_dt.columns[5]
print 'Feature 1: ', wow_data_dt.columns[1]
```

```
Feature 14: Race_Troll
Feature 1:  Level_m
```

```
from pyspark.mllib.evaluation import MulticlassMetrics

def getPredictionsLabels(model, test_data):
    predictions = model.predict(test_data.map(lambda r: r.features))
    return predictions.zip(test_data.map(lambda r: r.label))

def printMetrics(predictions_and_labels):
    metrics = MulticlassMetrics(predictions_and_labels)
    print 'Precision of True ', metrics.precision(1)
    print 'Precision of False', metrics.precision(0)
    print 'Recall of True    ', metrics.recall(1)
    print 'Recall of False   ', metrics.recall(0)
    print 'F-1 Score         ', metrics.fMeasure()
    print 'Confusion Matrix\n', metrics.confusionMatrix().toArray()

predictions_and_labels = getPredictionsLabels(model, testing_data)

printMetrics(predictions_and_labels)
```

```
Precision of True  0.927823050058
Precision of False 0.0
Recall of True     1.0
Recall of False    0.0
F-1 Score          0.927823050058
```

```
Confusion Matrix
[[    0.   372.]
 [    0.  4782.]]
```

The overall accuracy, ie F-1 score, seems quite good, but one troubling issue is the discrepancy between the recall measures. The recall (aka sensitivity) for the Churn=True samples is high, while the recall for the Churn=False examples is relatively low. Perhaps the model's sensitivity bias toward Churn=True samples is due to a skewed distribution of the two types of samples. The confusion matrix Actual False Vs Predicted False also doesn't give result. Not a great confusion matrix to actually look at.

Applying Random Forest Algorithm

```
wow_data_rf = wow_data.select("Level_l","Level_m","Level_h","Race_Orc","Race_Tauren","Race_Troll","Race_Undead","Class_Warrior",
"Class_Hunter","Class_Rogue",
                              "Class_Shaman","Class_Warlock","Class_Druid","Class_Mage","Class_Priest","Churn")
```

```
from pyspark.ml.classification import RandomForestClassifier as RF
from pyspark.ml.feature import StringIndexer, VectorIndexer, VectorAssembler, SQLTransformer

from pyspark.ml import Pipeline
from pyspark.ml.evaluation import MulticlassClassificationEvaluator, BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
import numpy as np
import functools
#from pyspark.ml.feature import OneHotEncoder

cols_now = ["Level_l","Level_m","Level_h","Race_Orc","Race_Tauren","Race_Troll","Race_Undead","Class_Warrior","Class_Hunter","Cl
ass_Rogue",
                              "Class_Shaman","Class_Warlock","Class_Druid","Class_Mage","Class_Priest"]

assembler_features = VectorAssembler(inputCols=cols_now, outputCol='features')
labelIndexer = StringIndexer(inputCol="Churn", outputCol='label').fit(wow_data_rf)
tmp = [assembler_features, labelIndexer]
pipeline = Pipeline(stages=tmp)
```

```
allData = pipeline.fit(wow_data_rf).transform(wow_data_rf)
allData.cache()
trainingData, testData = allData.randomSplit([0.70,0.30], seed=0)
```

```
rf = RF(labelCol='label', featuresCol='features',numTrees=150)
fit = rf.fit(trainingData)
transformed = fit.transform(testData)
```

```
transformed.select("Churn", "probability", "prediction").show(5)
```

```
+-----+--------------------+----------+
|Churn|         probability|prediction|
+-----+--------------------+----------+
|    0|[0.79257114730507...|       0.0|
|    0|[0.79257114730507...|       0.0|
|    0|[0.79257114730507...|       0.0|
|    0|[0.79257114730507...|       0.0|
|    0|[0.79257114730507...|       0.0|
+-----+--------------------+----------+
only showing top 5 rows
```

```
from pyspark.mllib.evaluation import BinaryClassificationMetrics as metric
results = transformed.select(['probability', 'label'])

results_collect = results.collect()
results_list = [(float(i[0][0]), 1.0-float(i[1])) for i in results_collect]
scoreAndLabels = sc.parallelize(results_list)

metrics = metric(scoreAndLabels)
print("Accuracy: ", metrics.areaUnderROC)
```

```
('Accuracy: ', 0.7337465418432121)
```

Applying cross validation

```
wow_data_cv = wow_data.select("Level_l","Level_m","Level_h","Race_Orc","Race_Tauren","Race_Troll","Race_Undead","Class_Warrior",
"Class_Hunter","Class_Rogue",
                              "Class_Shaman","Class_Warlock","Class_Druid","Class_Mage","Class_Priest","Churn")
```

Changing the data ratio

```
stratified_CV_data = wow_data_cv.sampleBy('Churn', fractions={0: 1.0, 1: .10}).cache()

stratified_CV_data.groupby('Churn').count().toPandas()
```

	Churn	count
0	1	2466
1	0	1906

```
training_data, testing_data = labelData(stratified_CV_data).randomSplit([0.8, 0.2])
```

The ML package supports k-fold cross validation, which can be readily coupled with a parameter grid builder and an evaluator to construct a model selection workflow. Below, we'll use a transformation/estimation pipeline to train our models. The cross validator will use the ParamGridBuilder to iterate through the maxDepth parameter of the decision tree and evaluate the models using the F1-score, repeating 3 times per parameter value for reliable results.

```
# Train a DecisionTree model
dTree = DecisionTreeClassifier(labelCol='indexedLabel', featuresCol='indexedFeatures')

# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, dTree])

# Search through decision tree's maxDepth parameter for best model
paramGrid = ParamGridBuilder().addGrid(dTree.maxDepth, [2,3,4,5,6,7]).build()

# Set F-1 score as evaluation metric for best model selection
evaluator = MulticlassClassificationEvaluator(labelCol='indexedLabel',
                                              predictionCol='prediction', metricName='f1')

# Set up 3-fold cross validation
crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=evaluator,
                          numFolds=3)

CV_model = crossval.fit(vectorized_CV_data)

# Fetch best model
tree_model = CV_model.bestModel.stages[2]
print tree_model
```

```
DecisionTreeClassificationModel (uid=DecisionTreeClassifier_4fc4b13413dfdaf81827) of depth 4 with 31 nodes
```

We find that the best tree model produced using the cross-validation process is one with a depth of 4. So we can assume that our initial "shallow" tree of depth 2 in the previous section was not complex enough, while trees of depth higher than 4 overfit the data and will not perform well in practice.

The actual performance of the model can be determined using the final_test_data set which has not been used for any training or cross-validation activities. We'll transform the test set with the model pipeline, which will map the labels and features according to the same recipe. The evaluator will provide us with the F-1 score of the predictions, and then we'll print them along with their probabilities. Predictions on new, unlabeled customer activity data can also be made using the same pipeline CV_model.transform() function.

```
final_test_data =  wow_data.select("Level_l","Level_m","Level_h","Race_Orc","Race_Tauren","Race_Troll","Race_Undead","Class_Warr
ior","Class_Hunter","Class_Rogue",
                            "Class_Shaman","Class_Warlock","Class_Druid","Class_Mage","Class_Priest","Churn")
```

```
vectorized_test_data = vectorizeData(final_test_data)

transformed_data = CV_model.transform(vectorized_test_data)
print evaluator.getMetricName(), 'accuracy:', evaluator.evaluate(transformed_data)

predictions = transformed_data.select('indexedLabel', 'prediction', 'probability')
predictions.toPandas().head()
```

```
f1 accuracy: 0.835710553138
```

	indexedLabel	prediction	probability
0	0.0	1.0	[0.317757009346, 0.682242990654]
1	0.0	1.0	[0.224890829694, 0.775109170306]
2	0.0	1.0	[0.26, 0.74]
3	0.0	0.0	[0.785912882298, 0.214087117702]
4	0.0	0.0	[0.785912882298, 0.214087117702]

The prediction probabilities can be very useful in ranking customers by their likeliness to defect. This way, the limited resources available to the business for retention can be focused on the appropriate customers.

5.6 CHURN PREDICTION – METHOD 2 ON FEATURE SET 2

Apply Random Forest Classifier from SparkML

```
from pyspark.ml.classification import RandomForestClassifier as RF
from pyspark.ml.feature import StringIndexer, VectorIndexer, VectorAssembler, SQLTransformer

from pyspark.ml import Pipeline
from pyspark.ml.evaluation import MulticlassClassificationEvaluator, BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
import numpy as np
import functools
from pyspark.ml.feature import OneHotEncoder

cols_now = ['Class_Warlock','Class_Druid', 'Class_Hunter',
            'Class_DeathKnight', 'Class_Paladin', 'Class_Rogue',
            'Class_Mage', 'Class_Priest', 'Class_Warrior',
            'Class_Shaman', 'Race_Orc', 'Race_Tauren', 'Race_Undead',
            'Race_BloodElf', 'Race_Troll', 'ZonesPlayed', 'LevelFlag',
            'GuildFlag', 'DaysPlayed', 'PlayerTenure'
]

assembler_features = VectorAssembler(inputCols=cols_now, outputCol='features')
```

```
labelIndexer = StringIndexer(inputCol="Churn", outputCol='label').fit(churndf)
tmp = [assembler_features, labelIndexer]
pipeline = Pipeline(stages=tmp)
```

```
allData = pipeline.fit(churndf).transform(churndf)
allData.cache()

trainingData, testData = allData.randomSplit([0.70,0.30], seed=0)
```

```
labelIndexer = StringIndexer(inputCol="Churn", outputCol='label').fit(churndf)
tmp = [assembler_features, labelIndexer]
pipeline = Pipeline(stages=tmp)
```

```
allData = pipeline.fit(churndf).transform(churndf)
allData.cache()

trainingData, testData = allData.randomSplit([0.70,0.30], seed=0)
```

```
rf = RF(labelCol='label', featuresCol='features',numTrees=150)
fit = rf.fit(trainingData)
transformed = fit.transform(testData)
```

```
transformed.select("Churn", "probability", "prediction").show(5)
```

```
+-----+--------------------+----------+
|Churn|         probability|prediction|
+-----+--------------------+----------+
|    1|[0.36603045585054...|       1.0|
|    1|[0.73221408595827...|       0.0|
|    1|[0.27061122241881...|       1.0|
|    1|[0.66975419278734...|       0.0|
|    1|[0.65425018364164...|       0.0|
+-----+--------------------+----------+
only showing top 5 rows
```

Calculate Model Accuracy

```
from pyspark.mllib.evaluation import BinaryClassificationMetrics as metric
results = transformed.select(['probability', 'label'])

results_collect = results.collect()
results_list = [(float(i[0][0]), 1.0-float(i[1])) for i in results_collect]
scoreAndLabels = sc.parallelize(results_list)

metrics = metric(scoreAndLabels)
print("Accuracy: ", metrics.areaUnderROC)
```

```
('Accuracy: ', 0.8102501007455151)
```

Apply DecisionTree Classifier

```
from pyspark.ml.classification import DecisionTreeClassifier as DF
```

```
dt = DF(maxDepth=6, labelCol="label")
```

```
model = dt.fit(trainingData)
```

```
transformed = model.transform(testData)
```

```
transformed.select("Churn", "probability", "prediction").show(5)
```

```
+-----+--------------------+----------+
|Churn|         probability|prediction|
+-----+--------------------+----------+
|    1|[0.22644497228820...|       1.0|
|    1|[0.69613259668508...|       0.0|
|    1|[0.30381383322559...|       1.0|
|    1|[0.31547619047619...|       1.0|
|    1|[0.59923664122137...|       0.0|
+-----+--------------------+----------+
only showing top 5 rows
```

```
from pyspark.mllib.evaluation import BinaryClassificationMetrics as metric
results = transformed.select(['probability', 'label'])

results_collect = results.collect()
results_list = [(float(i[0][0]), 1.0-float(i[1])) for i in results_collect]
scoreAndLabels = sc.parallelize(results_list)

metrics = metric(scoreAndLabels)
print("Accuracy: ", metrics.areaUnderROC)
```

```
('Accuracy: ', 0.8145357030917678)
```

Optimize model with Variable of Importance

Optimize Random Forest

```
cols_now = ['LevelFlag','Class_DeathKnight','DaysPlayed','PlayerTenure','ZonesPlayed']
assembler_features = VectorAssembler(inputCols=cols_now, outputCol='features')
labelIndexer = StringIndexer(inputCol="Churn", outputCol='label').fit(churndf)
tmp = [assembler_features, labelIndexer]
pipeline = Pipeline(stages=tmp)
```

```
rf = RF(labelCol='label', featuresCol='features',numTrees=30)
fit = rf.fit(trainingData)
transformed = fit.transform(testData)
```

```
transformed.select("Churn", "probability", "prediction").show(5)
```

```
+-----+--------------------+----------+
|Churn|         probability|prediction|
+-----+--------------------+----------+
|    1|[0.39254122194260...|       1.0|
|    1|[0.71429660591830...|       0.0|
|    1|[0.26160624572436...|       1.0|
|    1|[0.63316103479664...|       0.0|
|    1|[0.60617262348066...|       0.0|
+-----+--------------------+----------+
only showing top 5 rows
```

```
results = transformed.select(['probability', 'label'])

results_collect = results.collect()
results_list = [(float(i[0][0]), 1.0-float(i[1])) for i in results_collect]
scoreAndLabels = sc.parallelize(results_list)

metrics = metric(scoreAndLabels)
print("Accuracy: ", metrics.areaUnderROC)
```

```
('Accuracy: ', 0.8069571775583774)
```

```
transformed.crosstab('Churn', 'prediction').show()
```

```
+----------------+----+----+
|Churn_prediction| 0.0| 1.0|
+----------------+----+----+
|               1| 559|3641|
|               0|2752|1502|
+----------------+----+----+
```

Optimize DecisionTree

```
dt = DF(maxDepth=10, labelCol="label")
model = dt.fit(trainingData)
transformed = model.transform(testData)
transformed.select("Churn", "probability", "prediction").show(5)
```

```
+-----+--------------------+----------+
|Churn|         probability|prediction|
+-----+--------------------+----------+
|    1|[0.28158844765342...|       1.0|
|    1|[0.71428571428571...|       0.0|
|    1|[0.18734793187347...|       1.0|
|    1|           [0.0,1.0]|       1.0|
|    1|[0.68604651162790...|       0.0|
+-----+--------------------+----------+
only showing top 5 rows
```

```
results = transformed.select(['probability', 'label'])

results_collect = results.collect()
results_list = [(float(i[0][0]), 1.0-float(i[1])) for i in results_collect]
scoreAndLabels = sc.parallelize(results_list)

metrics = metric(scoreAndLabels)
print("Accuracy: ", metrics.areaUnderROC)
```

```
('Accuracy: ', 0.8152905108917098)
```

```
transformed.crosstab('Churn', 'prediction').show()
```

```
+----------------+----+----+
|Churn_prediction| 0.0| 1.0|
+----------------+----+----+
|               1| 488|3712|
|               0|2763|1491|
+----------------+----+----+
```

Gradient-boosted tree classifier

```
from pyspark.ml.classification import GBTClassifier


# Index labels, adding metadata to the label column.
# Fit on whole dataset to include all labels in index.
labelIndexer = StringIndexer(inputCol="Churn", outputCol="indexedLabel").fit(churndf)
# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as continuous.
cols_now = ['LevelFlag','Class_DeathKnight','DaysPlayed','PlayerTenure']
featureIndexer =    VectorAssembler(inputCols=cols_now, outputCol="indexedFeatures")
```

```
# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = churndf.randomSplit([0.7, 0.3])
```

```
# Train a GBT model.
gbt = GBTClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures", maxIter=10)
```

```
# Chain indexers and GBT in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, gbt])
```

```
# Train model.  This also runs the indexers.
model = pipeline.fit(trainingData)
```

```
# Make predictions.
predictions = model.transform(testData)
```

```
# Select example rows to display.
predictions.select("prediction", "indexedLabel").show(5)
```

```
+----------+------------+
|prediction|indexedLabel|
+----------+------------+
|       0.0|         1.0|
|       0.0|         1.0|
|       0.0|         1.0|
|       0.0|         1.0|
|       0.0|         1.0|
+----------+------------+
only showing top 5 rows
```

```
results = predictions.select(['probability', 'indexedLabel','prediction'])

results_collect = results.collect()
results_list = [(float(i[0][0]), 1.0-float(i[1])) for i in results_collect]
scoreAndLabels = sc.parallelize(results_list)

metrics = metric(scoreAndLabels)
print("Accuracy: ", metrics.areaUnderROC)
```

```
('Accuracy: ', 0.7877093775804722)
```

```
 predictions.crosstab('Churn','prediction').show()
```

```
+----------------+----+----+
|Churn_prediction| 0.0| 1.0|
+----------------+----+----+
|               1| 606|3703|
|               0|2654|1524|
+----------------+----+----+
```

```
gbtModel = model.stages[2]
print(gbtModel)  # summary only
```

```
GBTClassificationModel (uid=GBTClassifier_476bbbd40601071488b6) with 10 trees
```

DecisionTree with or without regularization gives the better training accuracy of 81% and test accuracy of 70% compared to Random Forest classifier and Gradient-boosted tree classifier.
The values of False Positive and False Negative are also lesser compared to other two models.

5.7 RECOMMENDATION ENGINE – KNN AND USER BASED COLLABORATIVE ENGINE – FEATURE SET 1

Change the Dataframe to User Item matrix and then a Similarity matrix.

A distance metric commonly used in recommender systems is cosine similarity, where the ratings are seen as vectors in n-dimensional space and the is calculated based on the angle between these vectors.

```
wide_df_g = df_g.pivot(index = 'Guild', columns = 'AvatarID', values = 'PatternCount').fillna(0)
wide_df_rc = df_rc_final.pivot(index = 'RaceClass', columns = 'AvatarID', values = 'RaceClassCount').fillna(0)
```

```
wide_df_rc_sparse = csr_matrix(wide_df_rc.values)
wide_df_g_sparse = csr_matrix(wide_df_g.values)
```

Create a KNN model Cosine matric sepeartely for Race & Class and Guild recommenders fit the models to the similarity matirx we created.

```
model_rc = NearestNeighbors(metric = 'cosine', algorithm = 'brute')
model_g = NearestNeighbors(metric = 'cosine', algorithm = 'brute')
model_rc.fit(wide_df_rc_sparse)
model_g.fit(wide_df_g_sparse)
```

```
NearestNeighbors(algorithm='brute', leaf_size=30, metric='cosine',
         metric_params=None, n_jobs=1, n_neighbors=5, p=2, radius=1.0)
```

Create 2 sepearte functions for Race & Class and Guild recommenders.

For ramdomly queried index we will get the nearest neighbors from the above KNN model and recommend to User with the randomly queried index.

```
def raceclass_recommender(wide_df_rc_sparse,wide_df_rc):

    query_index = np.random.choice(wide_df_rc.shape[0])
    print ('Hello {0}!! \n'.format(wide_df_rc.columns[query_index]))
    distances, indices = model_rc.kneighbors(wide_df_rc.iloc[query_index, :].reshape(1, -1), n_neighbors = 6)

    for i in range(0, len(distances.flatten())):
        if i == 0:
            print ('Below are recommendations for Race and Class combinations:\n')
        else:
            print ('{0}: {1}, with distance of {2}:'.format(i, wide_df_rc.index[indices.flatten()[i]], distances.flatten()[i]))
```

```
def guild_recommender(wide_df_g_sparse,wide_df_g):

    query_index_g = np.random.choice(wide_df_g.shape[0])
    print ('Hello {0}!! \n'.format(wide_df_g.columns[query_index_g]))
    distances_g, indices_g = model_g.kneighbors(wide_df_g.iloc[query_index_g, :].reshape(1, -1), n_neighbors = 6)

    for i in range(0, len(distances_g.flatten())):
        if i == 0:
            print ('Below are new Guild Recommendations:\n')
        else:
            print ('{0}: {1} Guild, with distance of {2}:'.format(i, wide_df_g.index[indices_g.flatten()[i]], distances_g.flatte
n()[i]))
```

Recommend Guilds for new users or users who are not part of any Guilds.

For a randomly selected User with no Guild, take the Level. Take the Guilds of the same Level with top Pattern Counts, as they are the Users with Guilds who played more. Recommend those to our User!!

```
def guild_recommender_new_users(df_no_g,df_g):

    query_index_no_g = np.random.choice(df_no_g.shape[0])
    query_user = df_no_g.iloc[query_index_no_g, :]
    df_rec = df_g[df_g['Level'].isin([query_user[1]])]
    df_rec=df_rec.sort_values(['PatternCount'], ascending=False).head()
    df_rec = df_rec[['Guild','PatternCount']]
    df_rec=df_rec.to_dict(orient='list')
    print ('Hello {0}!! \n'.format(query_user[0]))
    print ('Below are your Guild Recommendations:\n')
    print (df_rec['Guild'])
```

```
raceclass_recommender(wide_df_rc_sparse, wide_df_rc)
```

```
Hello 7!!

Below are recommendations for Race and Class combinations:

1: Undead & Rogue, with distance of 0.9999242420478779:
2: Blood Elf & Hunter, with distance of 0.9999430869321557:
3: Undead & Mage, with distance of 0.9999452786632852:
4: Tauren & Warrior, with distance of 0.9999789444668533:
5: Blood Elf & Paladin, with distance of 0.9999911328926738:
```

```
guild_recommender(wide_df_g_sparse, wide_df_g)
```

```
Hello 109!!

Below are new Guild Recommendations:

1: 235 Guild, with distance of 0.22721195199458677:
2: 55 Guild, with distance of 0.5781964490304243:
3: 6 Guild, with distance of 0.8795782867505325:
4: 0 Guild, with distance of 0.9283130820051884:
5: 201 Guild, with distance of 0.9362782951157229:
```

```
guild_recommender_new_users(df_no_g,df_g)
```

```
Hello 72885!!

Below are your Guild Recommendations:

['65', '220', '36', '24', '53']
```

As in when needed or as required we can recommend one of the below or all in combination to personalize Users game playing experience

Race Class combinations New Guilds for users already playing in Guilds Guilds for new Users or Users not playing in Guilds

Collaborative filtering produces recommendations based on the knowledge of users' attitude to items, that is it uses the "wisdom of the crowd" to recommend items.

In general, Collaborative filtering is the workhorse of recommender engines.

5.8 RECOMMENDATION ENGINE – KNN AND USER BASED COLLABORATIVE ENGINE – FEATURE SET 2

Build Recommendation Model - userbased collaboarative filtering

```
class GuildRecommendar:
      #Remove duplicate records
        def remove_duplicates(self,inputdata):
            initial_rows = inputdata.shape[0]
            print('Initial dataframe shape {0}'.format(inputdata.shape))
            inputdata = inputdata.drop_duplicates(['AvatarID', 'Guild'])
            current_rows = inputdata.shape[0]
            print('New dataframe shape {0}'.format(inputdata.shape))
            print('Removed {0} rows'.format(initial_rows - current_rows))

        def recommend_Guild(self,usertdataset,guilddataset,user_index,neighbors):
            from scipy.sparse import csr_matrix
            from sklearn.neighbors import NearestNeighbors
            wide_user_data_less_sparse = csr_matrix(usertdataset.values)
            model_knn = NearestNeighbors(metric = 'cosine', algorithm = 'brute')
            model_knn.fit(wide_user_data_less_sparse)
            distances, indices = model_knn.kneighbors(usertdataset.iloc[user_index, :].values.reshape(1, -1), n_neighbors = neig
hbors)
```

```
      total = []
      for i in usertdataset.columns:
          for j in range(1,neighbors):
              rowname = usertdataset.iloc[indices[0][j]].name
              if usertdataset.loc[usertdataset.iloc[indices[0][0]].name][i] == 0.0:
                  if usertdataset.loc[rowname][i] > 0.0:
                      totalplay = guilddataset.loc[guilddataset.Guild == i, 'GuildGamesPlayed'].values
                      weight = ((1/distances[0][j])*usertdataset.loc[rowname][i])/totalplay
                      total.append({'guild': i, 'weight': weight})
      df = pd.DataFrame(total)
      df.drop_duplicates(['guild'],keep='first',inplace=True)
      df.weight = df.weight.astype(float)
      df = df.sort_values(['weight'],ascending = False)
      df = df[['guild']]
      return df

  #Pick a random user to provide as input in recommendation functions
  def pickrandomuser(self,usertdataset):
      import numpy as np
      query_index = np.random.choice(usertdataset.shape[0])
      return query_index

  #Pick a random guild to provide as input in recommendation functions
  def pickrandomguild(self,usertdataset):
      import numpy as np
      guildid = np.random.choice(usertdataset.shape[1])
      guild = usertdataset.columns[guildid]
      return guild

  #Execute this function to check if a user will play in the guild
  def willuserplay(self,user_index,guild,usertdataset,guilddataset,neighbors=6):
      df2 = self.recommend_Guild(usertdataset,guilddataset,user_index,neighbors)
      if (df2.guild==guild).any():
          print ('It is very likely that this user will play if he joins Guild: %s' % str(guild))
      else:
          print ('It is not likely that this user will play if he joins Guild: %s' % str(guild))
```

```
gurec = GuildRecommendar()
```

```
gurec.pickrandomguild(matrix)
```

```
u'268'
```

```
gurec.pickrandomuser(matrix)
```

```
4224
```

```
gurec.recommend_Guild(matrix,GuildGames,65436,6)
```

	guild
1	370
2	9
0	21

```
gurec.willuserplay(65436,'268',matrix,GuildGames,6)
```

```
It is not likely that this user will play if he joins Guild: 268
```

```
gurec.willuserplay(65436,'370',matrix,GuildGames,6)
```

```
It is very likely that this user will play if he joins Guild: 370
```

This recommendation engine will suggest a suitable guild for the user to join and play and thus avoid player churning.

V. MODEL EVALUATION

i. Forecast the number of players expected in future time point

Evalation parameter	Models Attempted	Best Score	Final Model	Code
MAPE $\left(\frac{1}{n}\sum\frac{\lvert Actual - Forecast\rvert}{\lvert Actual\rvert}\right)*100$ RMSE $\text{RMSE} = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2}$	Holt Winters	Forecasting MAPE: 0.65 RMSE: 1087.3	Holt Winters	3_01_TimeSeries_Forecasting_HoltWinters_s.ipynb
	ARIMA	Future 24 Periods Prediction MAPE: 51.5 RMSE: 2974.26		
	LSTM Tensowflow			

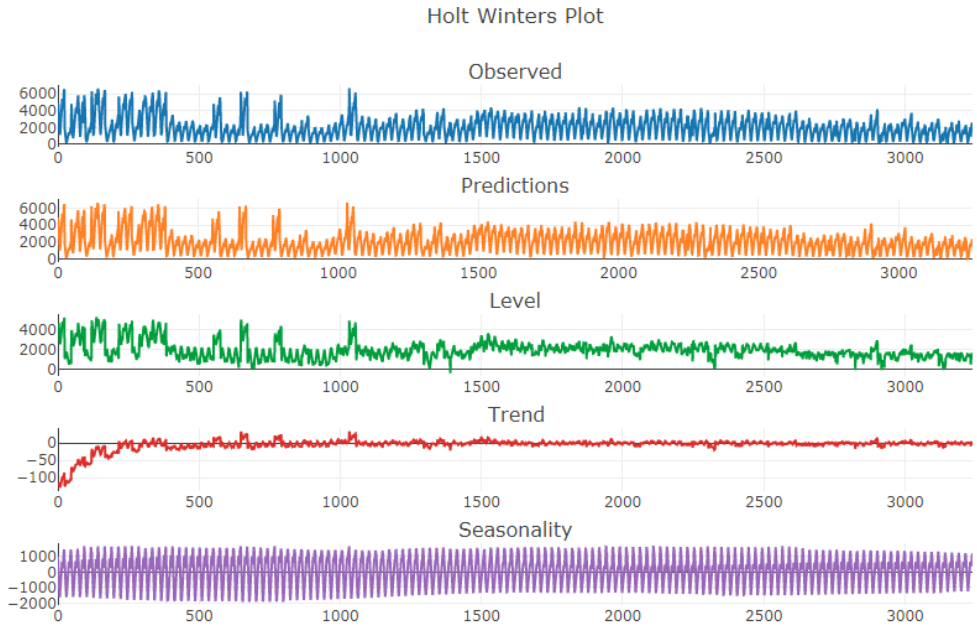
Note:
Have created a package sulekha_holtwinters to implement this model using Pyspark dataframes.
pip install Sulekha_holtwinters

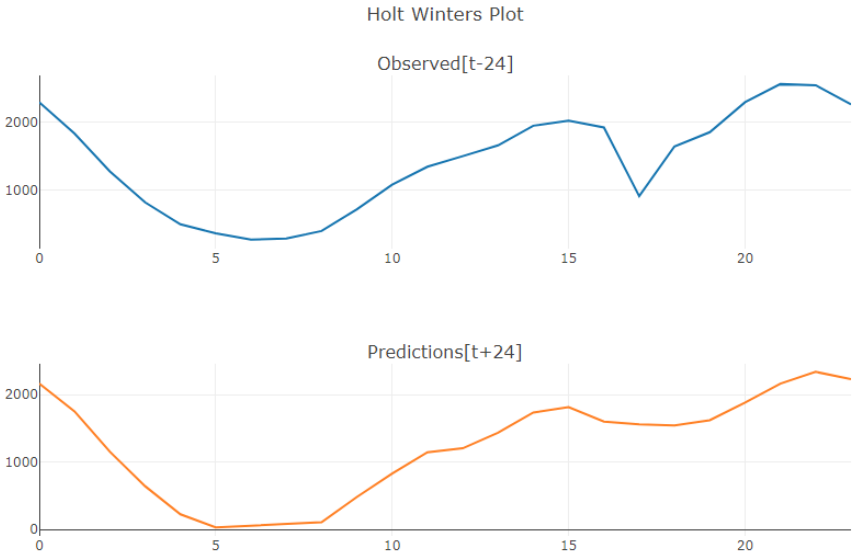
This RMSE and MAPE values are much lesser compared to the values from ARIMA model and LSTM Tensorflow attempted.

Hence Holt Winters Triple Exponential Smoothing - Additive method seems to be more dependable for wow dataset time series forecasting.

Accuracy of this model is highly dependent on the choice of values chosen for alpha, beta and gamma parameters. Even though there is a function available to calculate this value, choosing the optimal parameters is more based on trial and error.

Also, this model is more suitable for forecasting short term time points in future. If the model is attempted to predict for longer term periods in future, accuracy would be very less and values might just show a linear trend since historical data for these kind of games would become stale as the time increases.





HOLT-WINTERS SEASONAL METHOD

The Holt-Winters seasonal method comprises the forecast equation and three smoothing equations — one for the level ℓ_t, one for trend b_t, and one for the seasonal component denoted by s_t, with smoothing parameters α, β^* and γ. We use m to denote the period of the seasonality, i.e., the number of seasons in a year. For example, for quarterly data $m = 4$, and for monthly data $m = 12$.

The basic equations for their method are given by:

$$S_t = \alpha \frac{y_t}{I_{t-L}} + (1-\alpha)(S_{t-1} + b_{t-1}) \qquad \text{OVERALL SMOOTHING}$$

$$b_t = \gamma(S_t - S_{t-1}) + (1-\gamma)b_{t-1} \qquad \text{TREND SMOOTHING}$$

$$I_t = \beta \frac{y_t}{S_t} + (1-\beta)I_{t-L} \qquad \text{SEASONAL SMOOTHING}$$

$$F_{t+m} = (S_t + mb_t)I_{t-L+m} \qquad \text{FORECAST},$$

where

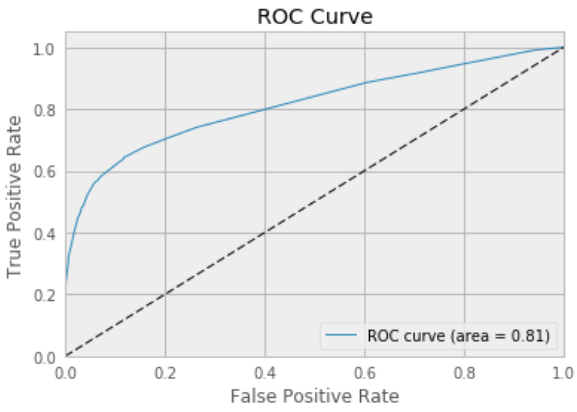
- y is the observation
- S is the smoothed observation
- b is the trend factor
- I is the seasonal index
- F is the forecast at m periods ahead
- t is an index denoting a time period

Source: http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc435.htm

ii. Predict player churning

Evalation parameter	Models Attempted	Best Score	Final Model	Code
$Accuracy = \frac{T_p + T_n}{T_p + T_n + F_p + F_n}$ $Precision = \frac{T_p}{T_p + F_p}$ $Recall = \frac{T_p}{T_p + T_n}$	Decistion Tree Classifier	Accuracy: 0.81	Decistion Tree Classifier	Featureset1 - 4_02_analysis_wow_data_d.ipynb Featureset2- 4_04_churnpredictio n_s.ipynb
	Random Forest Classifier	Precision: 0.88 Recall: 0.57		
	Gradient-boosted tree classifier	+----------+----+----+ |Churn_prediction| 0.0| 1.0| +----------+----+----+ | 1| 488|3712| | 0|2763|1491| +----------+----+----+		

Note:
Have performed an extensive feature engineering and created 22 new features from existing 7 features available in the dataset.



DECISION TREE CLASSIFIER

- Using the decision algorithm, we start at the tree root and split the data on the feature that results in the largest **information gain** (**IG**) (reduction in uncertainty towards the final decision).
- In an iterative process, we can then repeat this splitting procedure at each child node until the leaves are pure. This means that the samples at each leaf node all belong to the same class.
- In practice, we may set a limit on the depth of the tree to prevent overfitting. We compromise on purity here somewhat as the final leaves may still have some impurity.

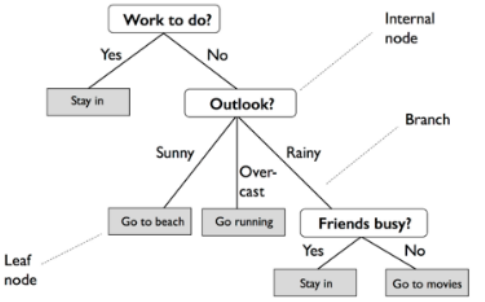
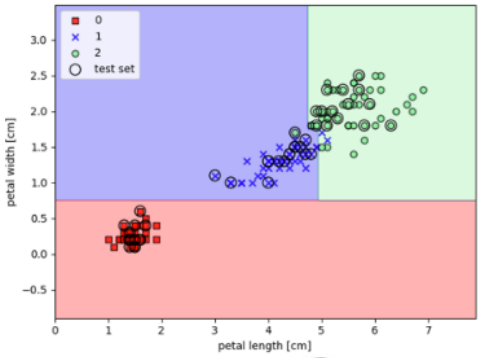
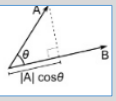


Image Credit : "Python Machine Learning" by Sebastian Raschka



iii. Recommend Guild, Race/Class to players

Models Applied	Calculation parameter	Code
K-Nearest Neighbors User based collaborative filtering	$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$ 	5_01_wow_recommender_engine_n.ipynb 5_02_recommendationengine_s.ipynb

Output:

```
Hello 109!!

Below are new Guild Recommendations:

1: 235 Guild, with distance of 0.22721195199458677:
2: 55 Guild, with distance of 0.5781964490304243:
3: 6 Guild, with distance of 0.8795782867505325:
4: 0 Guild, with distance of 0.9283130820051884:
5: 201 Guild, with distance of 0.9362782951157229:
```

```
Hello 7!!

Below are recommendations for Race and Class combinations:

1: Undead & Rogue, with distance of 0.9999242420478779:
2: Blood Elf & Hunter, with distance of 0.9999430869321557:
3: Undead & Mage, with distance of 0.9999452786632852:
4: Tauren & Warrior, with distance of 0.9999789444668533:
5: Blood Elf & Paladin, with distance of 0.9999911328926738:
```

KNN AND USER BASED COLLABORATIVE FILTERING

Example:

*K*NN and Collaborative Filtering

- **Collaborative Filtering Example**
  - A movie rating system
  - Ratings scale: 1 = "hate it"; 7 = "love it"
  - Historical DB of users includes ratings of movies by Sally, Bob, Chris, and Lynn
  - Karen is a new user who has rated 3 movies, but has not yet seen "Independence Day"; should we recommend it to her?
  - Approach: use kNN to find similar users, then combine their ratings to get prediction for Karen.

	Sally	Bob	Chris	Lynn	Karen
Star Wars	7	7	3	4	7
Jurassic Park	6	4	7	4	4
Terminator II	3	4	7	6	3
Independence Day	7	6	2	2	?

Will Karen like "Independence Day?"

Image credit: http://slideplayer.com/slide/4462580/

VI. COMPARISON TO BENCHMARK

Models attempted initially were initial benchmarks considered:

[Explained under the section: Appendix -> Models attempted before]

1. Neural Networks for players behavior pattern recognition
2. ARIMA for time series forecasting

Both the models did not provide expected results or accuracy.

COMPARISON OF FINAL SOLUTION TO INITIAL BENCHMARK

Highlevel Problem	Initial Benchmark	Problems Faced	Final Model	Advantages of Final Model
Players behavior Patterns Recognition	Neural Networks	1. The output was more of Exploratory Analysis on the data and there were no new findings on the players' behavior and hence it was not very useful to provide solutions to any business problem. 2. Model was more of a black box and hence was not easily explainable.	1. Decistion Tree for Churn Predictions 2. KNN and User based collaborative filtering for Recommendation	Model gave an accuracy of 81% with Precision at 88% and Recall at 57% which is more useful in identifying the customer Churn and based on their playing patterns, we came up with a Recommendation Engine to recommend them with appropriate Race, Class and Guild to minimize player churning
Time series Forecasting	ARIMA for daily data	1. Model captured the trend and errors from World of Warcraft dataset pretty well but there were daily and hourly seasonality in the data which was not captured by the model. 2. Mean Absolute percentage error was too high and the model was not very useful for Time series forecasting.	Holt Winters Forecasting Model	Model forecasting was performed on hourly data and every weekday's data was forecasted separately. MAPE for Forecasting: 0.65 RMSE for Forecasting: 1087.3 MAPE for Future 24 hours prediction: 51.5 RMSE for Future 24 hours prediction: 2974.26 This value is relatively accurate and would prove to be more useful in planning server and support people allocation on a daily basis.

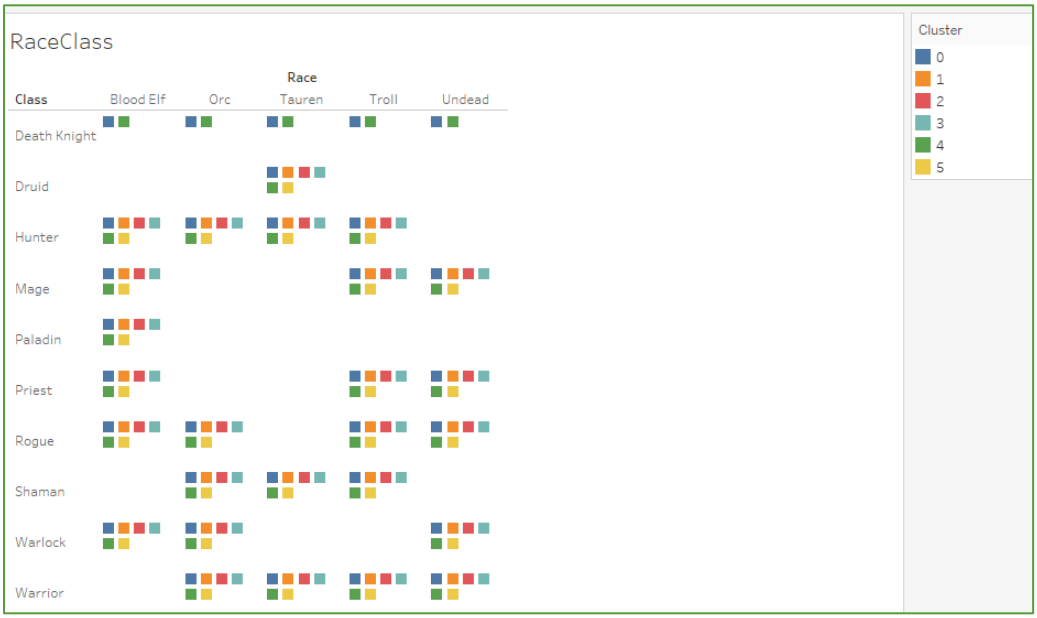
VII. VISUALIZATIONS

We have used Plotly and Tableau to perform visualizations on the dataset.

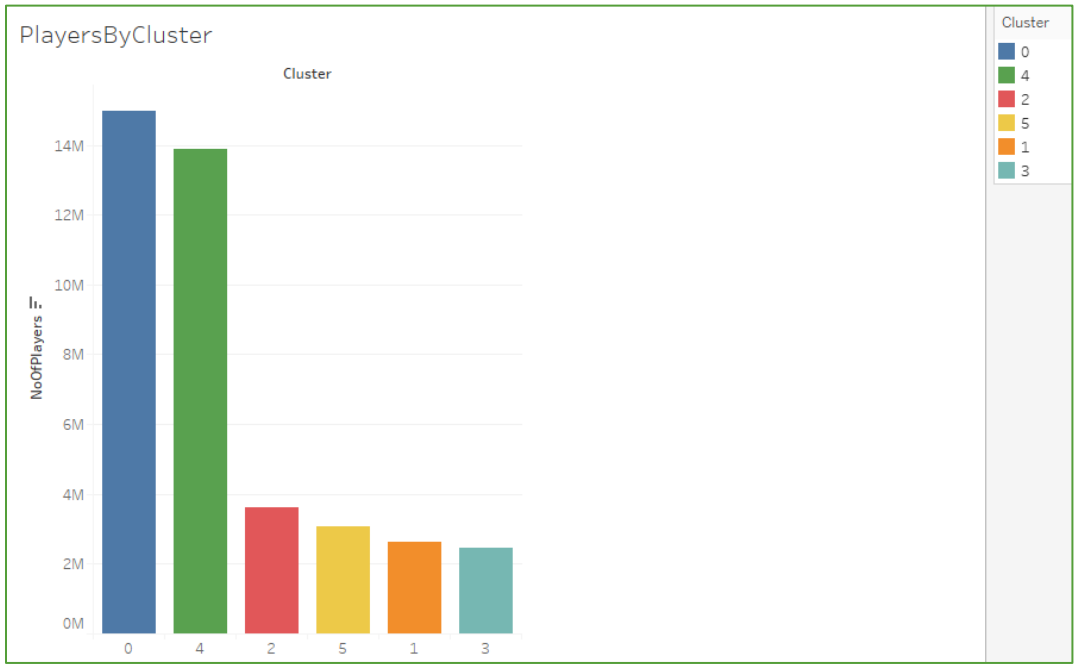
Exploratory visualization on the complete dataset is already provided under the above section.

Apart from the exploratory visualization of the input dataset, we have also performed visualizations on the intermediate datasets which helped us in proceeding with further model decisions.

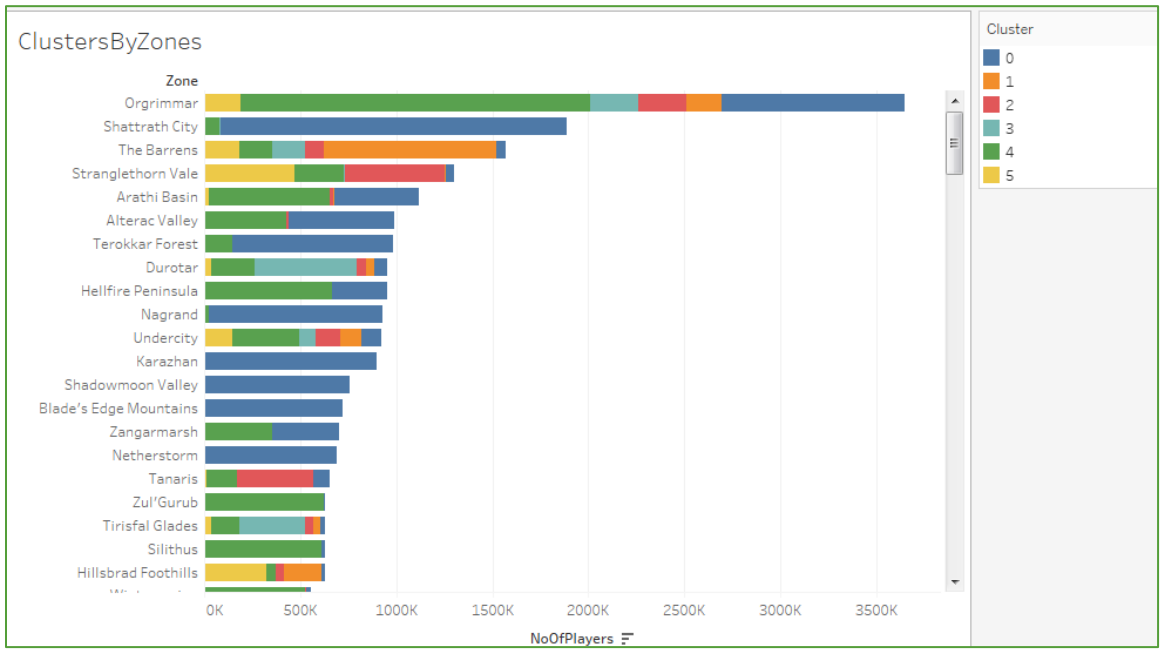
Cluster Data Analysis



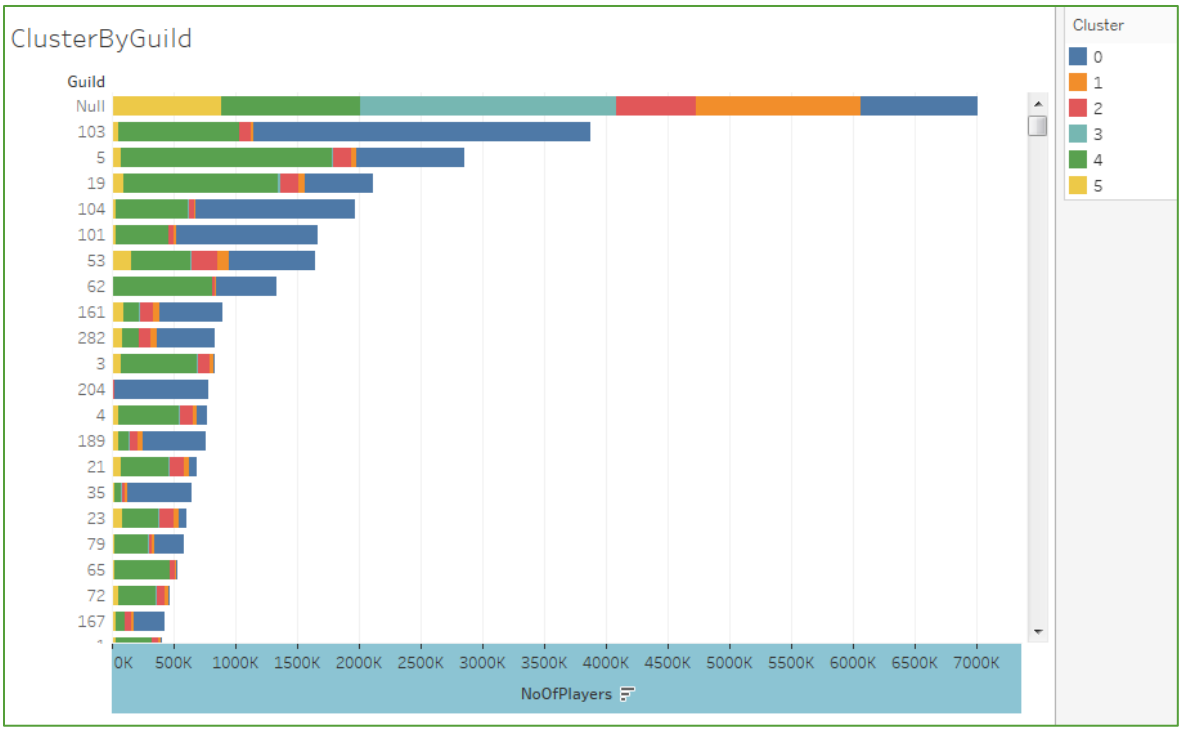
Warrior and Hunter Classes with a combination of BloodElf, Orc and Tauren races are played by maximum players



Majority of the players are grouped under Cluster 0



Origrimmar is the most played zone.

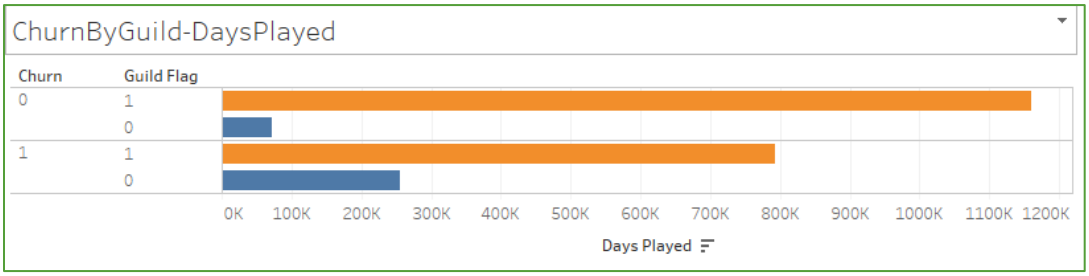


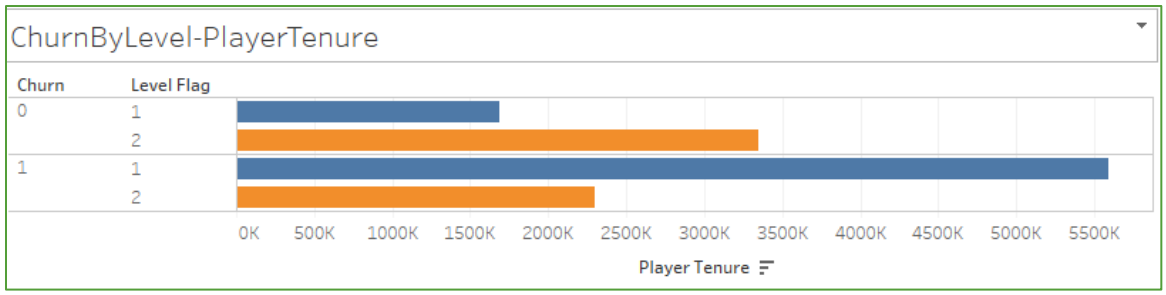
Maximum players played without Guild are sorted into Cluster 3

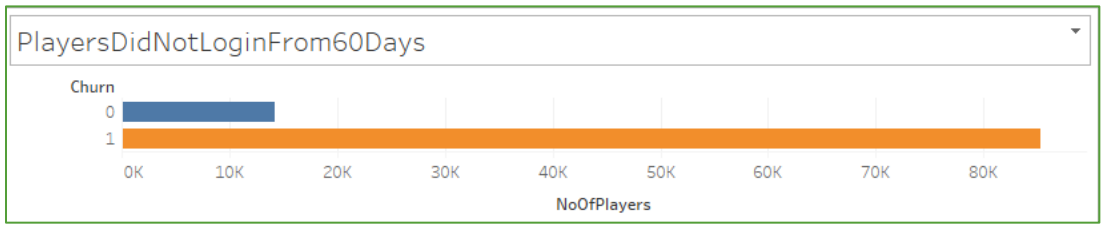
ClusterByRace

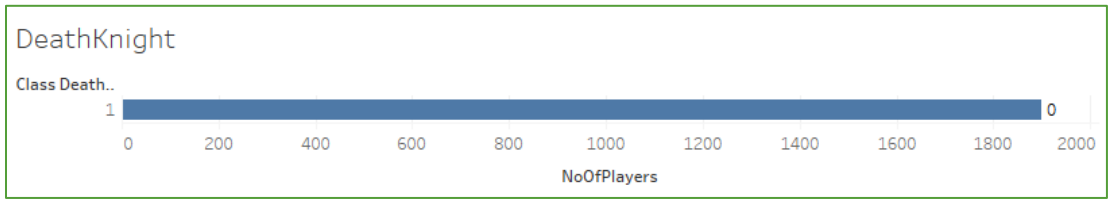
	Cluster					
Race	0	1	2	3	4	5
Undead	4,519,716	688,588	1,102,588	540,433	5,403,671	881,224
Tauren	3,409,419	496,311	765,734	399,185	3,330,764	606,095
Blood Elf	3,669,223	820,018	773,309	713,044	903,887	802,237
Troll	1,898,374	388,131	564,968	519,777	2,447,360	473,141
Orc	1,478,633	248,091	402,585	279,927	1,804,144	312,187

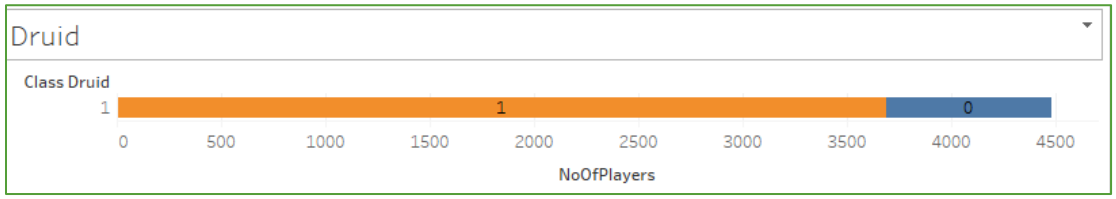
Churn Data Analysis

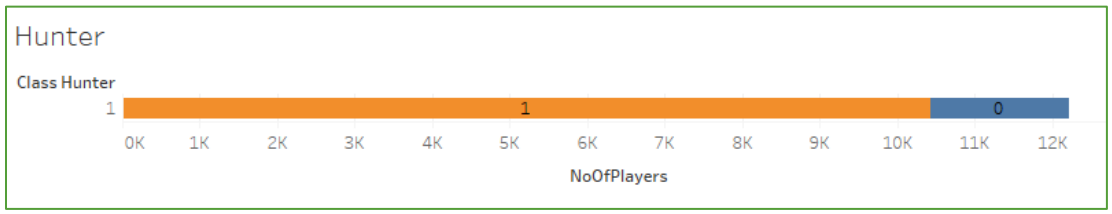


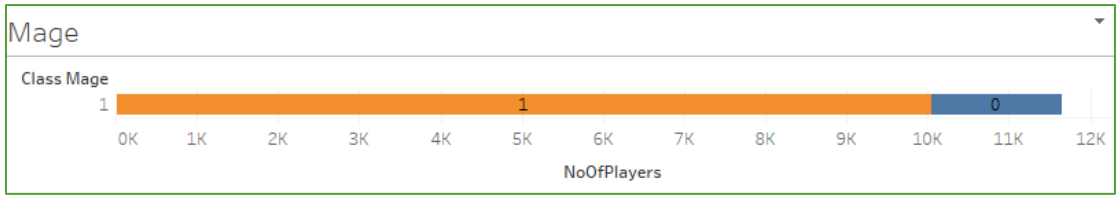

ChurnByLevel-PlayerTenure
Churn
Level Flag
0
1
2
1
1
2
0K
500K
1000K
1500K
2000K
2500K
3000K
3500K
4000K
4500K
5000K
5500K
Player Tenure

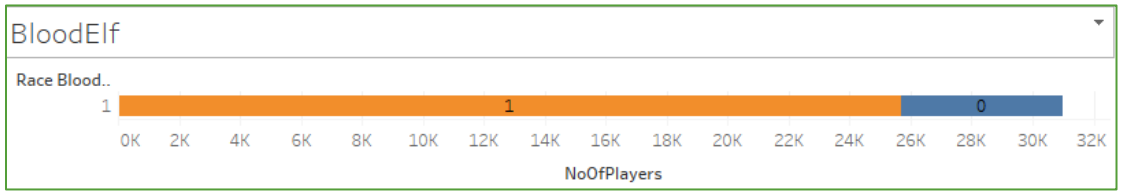

PlayersDidNotLoginFrom60Days
Churn
0
1
0K
10K
20K
30K
40K
50K
60K
70K
80K
NoOfPlayers

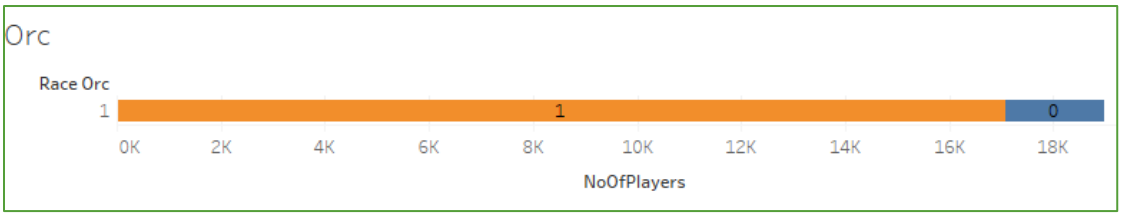

DeathKnight
Class Death..
1
0
0
200
400
600
800
1000
1200
1400
1600
1800
2000
NoOfPlayers


Druid
Class Druid
1
1
0
0
500
1000
1500
2000
2500
3000
3500
4000
4500
NoOfPlayers


Hunter
Class Hunter
1
1
0
0K
1K
2K
3K
4K
5K
6K
7K
8K
9K
10K
11K
12K
NoOfPlayers


Mage
Class Mage
1
1
0
0K
1K
2K
3K
4K
5K
6K
7K
8K
9K
10K
11K
12K
NoOfPlayers


BloodElf
Race Blood..
1
1
0
0K
2K
4K
6K
8K
10K
12K
14K
16K
18K
20K
22K
24K
26K
28K
30K
32K
NoOfPlayers



Insights

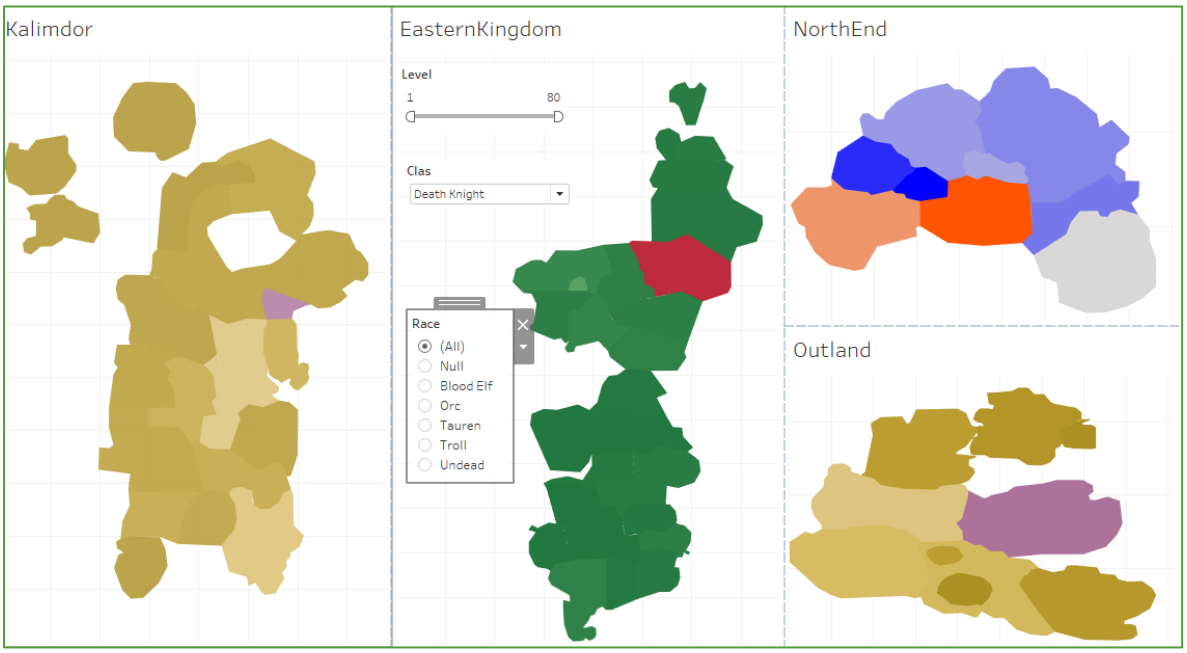
1. Churn is high when players were unable to go to next levels quickly.

Recommendation to Developers: Complexity of the initial levels needs to be brought down to let players play with more interest.

2. Interestingly Death Knight is the Class played by less number of Players but none of the players playing Death Knight have churned.

Recommendation to Developers: Recommend the Class “Death Knight” to more users so that they can explore and play more

WOW Zone Heat map



VIII. IMPLICATIONS

Insights from Prediction Models and Exploratory Analysis	Recommendations to Business
Hunter is the class chosen by most of the players	Develop classes similar to Hunter
Death Knight is chosen by least number of players but interestingly no player playing Death Knight Churned from 3 years	Recommend Death Knight to more users through Recommendation Engine by adding more weightage to it.
Undead is the Race chosen by most of the players	Recommend Undead combining with less played classes to more users.
Orc is chosen by least number of players. More players playing Orc have Churned	Improvise Orc to make it more interesting
Orgrimmar is the Zone chosen by most of the players	Orgimmar is one of the main cities with very interesting events, replicate similar events in other Zones too.
7,014,160 players are playing without joining any guilds. Players without Guilds	Recommend more active Guilds to players
January is the month with most players and October is with least players every year	Plan Server and IT support resource availability to be high during these months
Sunday and Saturday (Weekends) are the days with more players and Thursday is the day with least players every week	Plan Server and IT support resource availability to be high during these days based on the outcome of Holt Winters Model
10:00 PM is the most played hour in a day and 6:00 AM is the least played	Do not plan any maintenance or downtime during these time periods

Recommendations to Business are provided at a 95% confidence level

IX. LIMITATIONS

1. These models are derived based on the data available between the three year periods of 2006 to 2009.
2. In real world scenario: more recent data would be required to redesign this model and to maintain continuously.
3. Also, the features available in the dataset is very less (7 Features) which makes it challenging to provide more in depth insights and to identify more business problems and solutions.
4. To enhance the solutions provided by us, it would be very essential to capture logs with more information in future.

X. CLOSING REFLECTIONS

Learnings:

1. **Extensive usage of Tableau to bring more meaningful insights from World of Warcraft Logs**
2. **Multiple Time series Forecasting models and their application on WoW logs**
3. **Methods of application of Feature Engineering on WOW dataset since the available features were not enough of successful model building**
4. **Appropriate usage of Model Tuning and Regularization on WoW dataset**

Things to do differently next time:

1. **Collect data that has captured more features in future logs to bring more insights**
2. **Explore the usage of Spark R on this dataset**

XI. CODE INFORMATION

Input Data

http://mmnet.iis.sinica.edu.tw/dl/wowah/wowah.rar

Jupyter Notebook

Name	Date modified	Type	Size	Folder name
01_DataPrep_ExploratoryAnalysis	2/15/2018 8:26 PM	File folder		Codeset
02_Clustering	2/15/2018 8:27 PM	File folder		Codeset
03_TimeSeries	2/15/2018 8:27 PM	File folder		Codeset
04_ChurnPrediction	2/15/2018 8:27 PM	File folder		Codeset
05_Recommendation	2/15/2018 8:27 PM	File folder		Codeset

Name	Date modified	Type	Size	Folder name
1_01parse_wowlogs_s.ipynb	2/12/2018 9:56 AM	IPYNB File	1,510 KB	01_DataPrep_ExploratoryAnalysis
1_02_clean_data_alreadydone_s.ipynb	2/12/2018 10:00 AM	IPYNB File	18 KB	01_DataPrep_ExploratoryAnalysis
1_03_load_to_spark_s.ipynb	2/12/2018 10:01 AM	IPYNB File	5 KB	01_DataPrep_ExploratoryAnalysis
1_04_exploratoryDataAnalysis_Part1_s.ipynb	2/15/2018 8:21 PM	IPYNB File	33 KB	01_DataPrep_ExploratoryAnalysis
1_05_exploratoryDataAnalysis_Part2_s.ipynb	2/15/2018 8:21 PM	IPYNB File	38 KB	01_DataPrep_ExploratoryAnalysis

Name	Date modified	Type	Size	Folder name
2_01_spark_ml_wow_clusters_s.ipynb	2/15/2018 8:22 PM	IPYNB File	11 KB	02_Clustering

Name	Date modified	Type	Size	Folder name
3_01_TimeSeries_Forecasting_HoltWinters_s.ipynb	2/15/2018 8:22 PM	IPYNB File	6,783 KB	03_TimeSeries
3_02_Timeseries_Arima_s.ipynb	2/15/2018 8:22 PM	IPYNB File	1,148 KB	03_TimeSeries
3_03_Timeseries_Tensorflow_s.ipynb	2/15/2018 8:23 PM	IPYNB File	27 KB	03_TimeSeries

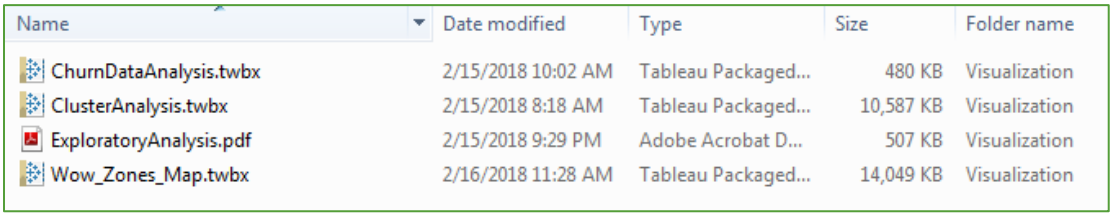
Name	Date modified	Type	Size	Folder name
4_01_wow_data_for_churn_create_d.ipynb	2/15/2018 8:23 PM	IPYNB File	30 KB	04_ChurnPrediction
4_02_analysis_wow_data_d.ipynb	2/15/2018 9:09 AM	IPYNB File	267 KB	04_ChurnPrediction
4_03_churn_featureengineering_s.ipynb	2/15/2018 8:24 PM	IPYNB File	36 KB	04_ChurnPrediction
4_04_churnprediction_s.ipynb	2/15/2018 8:24 PM	IPYNB File	259 KB	04_ChurnPrediction

Name	Date modified	Type	Size	Folder name
5_01_wow_recommender_engine_n.ipynb	2/15/2018 8:24 PM	IPYNB File	21 KB	05_Recommendation
5_02_recommendationengine_s.ipynb	2/15/2018 8:25 PM	IPYNB File	12 KB	05_Recommendation

HTML Files

Name	Date modified	Type	Size	Folder name
01_DataPrep_ExploratoryAnalysis	2/15/2018 8:25 PM	File folder		HTML_Codeset
02_Clustering	2/15/2018 8:26 PM	File folder		HTML_Codeset
03_TimeSeries	2/15/2018 8:26 PM	File folder		HTML_Codeset
04_ChurnPrediction	2/15/2018 8:26 PM	File folder		HTML_Codeset
05_Recommendation	2/15/2018 8:26 PM	File folder		HTML_Codeset

Tableau Visualizations



Name	Date modified	Type	Size	Folder name
ChurnDataAnalysis.twbx	2/15/2018 10:02 AM	Tableau Packaged...	480 KB	Visualization
ClusterAnalysis.twbx	2/15/2018 8:18 AM	Tableau Packaged...	10,587 KB	Visualization
ExploratoryAnalysis.pdf	2/15/2018 9:29 PM	Adobe Acrobat D...	507 KB	Visualization
Wow_Zones_Map.twbx	2/16/2018 11:28 AM	Tableau Packaged...	14,049 KB	Visualization

Project in Github with all of the above code:

https://github.com/capstoneaiscientists/Team3GLBDA

APPENDIX

Other models attempted before interim report

1. Neural networks for pattern recognition - Exploratory

In the process of solving problem statements, we first came up with identifying the game play patterns of players from the dataset.

For this process, we have converted the dataset into numerical data that can be provided as input to Pattern recognizing neural networks.

This model would still be part of exploratory data analysis.

Step 1: Read data into a dataframe

```
df2 = pd.read_csv("newlogs.csv", usecols = ['Guild','Level','Race','Class'])
```

Step 2: Group records by Guild, Level, Race and Class and count the number of records following each unique combination

```
df2 = df2.groupby(['Guild','Level','Race','Class']).size().reset_index()
```

```
df2.rename(columns = {0: 'PatternCount'},  inplace=True)
```

```
df2.head(5)
```

	Guild	Level	Race	Class	PatternCount
0		1	Blood Elf	Hunter	3834
1		1	Blood Elf	Mage	6621
2		1	Blood Elf	Paladin	10121
3		1	Blood Elf	Priest	4293
4		1	Blood Elf	Rogue	5787

Step 3: Convert string values on the above columns into their encoded numeric value

```
def addguildflag(x):
    if x== ' ':
        return 0
    else:
        return 1
```

```
df2['GuildFlag'] = df2.apply(lambda col: addguildflag(col['Guild']), axis = 1)
```

```
def addlevelflag(x):
    if x>=1 and x<=23:
        return 2
    elif x>=24 and x<=47:
        return 3
    else:
        return 4
```

```
df2['LevelFlag'] = df2.apply(lambda col: addlevelflag(col['Level']), axis = 1)
```

```
def addraceflag(x):
    if x == ' Blood Elf':
        return 5
    elif x == ' Orc':
        return 6
    elif x == ' Tauren':
        return 7
    elif x == ' Troll':
        return 8
    elif x == ' Undead':
        return 9
```

```
df2['RaceFlag'] = df2.apply(lambda col: addraceflag(col['Race']), axis = 1)
```

```
def addclassflag(x):
    if x in [' Warrior', ' Hunter', ' Rogue', ' Paladin', ' Death Knight']:
        return 10
    elif x in [' Shaman', ' Warlock', ' Druid', ' Mage', ' Priest']:
        return 11
```

Step 4: Create Frequency Flag by creating a rule for each frequency

```
a = df3.PatternCount.quantile(0.33)
b = df3.PatternCount.quantile(0.66)
```

```
def addfrequencyflag(x):
    if x>=1 and x<=a:
        return 18
    elif x>a and x<=b:
        return 19
    else:
        return 20
```

```
df3 = df2[['GuildFlag','LevelFlag','RaceFlag','ClassFlag','PatternCount']]
```

```
df3 = df3.groupby(by = ['GuildFlag','LevelFlag','RaceFlag','ClassFlag'])['PatternCount'].sum().reset_index()
```

```
df3['FrequencyFlag'] = df3.apply(lambda col: addfrequencyflag(col['PatternCount']), axis = 1)
```

Step 5: Create Pattern ID for each unique pattern

```
df3['PatternID'] = df3.index+1
```

```
df3
```

	GuildFlag	LevelFlag	RaceFlag	ClassFlag	PatternCount	FrequencyFlag	PatternID
0	0	2	5	10	516572	20	1
1	0	2	5	11	484618	20	2
2	0	2	6	10	257647	19	3
3	0	2	6	11	75869	18	4
4	0	2	7	10	217354	19	5
5	0	2	7	11	293998	19	6
6	0	2	8	10	420970	20	7
7	0	2	8	11	166371	18	8
8	0	2	9	10	235052	19	9
9	0	2	9	11	462959	20	10

Step 6: Identify the number of unique patterns in the dataset

```
import pandas as pd
import numpy as np
#cols = ['GuildFlag','LevelFlag','RaceFlag','ClassFlag','PatternID']
data = pd.read_csv("test1.csv")
data = data.sample(1000)
```

```
data.head(5)
```

	GuildFlag	LevelFlag	RaceFlag	ClassFlag	FrequencyFlag	PatternID
735	1	4	9	11	20	48
8099	1	4	9	11	20	48
31171	1	3	7	10	20	35
32234	1	3	9	10	20	39
44265	0	4	9	11	19	24

Step 7: Normalize the data and create Train & Test data

```
#Normalized
df_norm = data[['GuildFlag','LevelFlag','RaceFlag','ClassFlag']].apply(lambda x: (x - x.min()) / (x.max() - x.min()))
target = data[['PatternID']].apply(lambda x: (x - x.min()) / (x.max() - x.min()))
target.PatternID.nunique()
```

```
46
```

```
df = pd.concat([df_norm, target], axis=1)
df.sample(n=4)
```

	GuildFlag	LevelFlag	RaceFlag	ClassFlag	PatternID
21861	1.0	1.0	1.000000	1.0	1.000000
32025	1.0	0.5	0.333333	0.0	0.723404
33893	1.0	1.0	0.000000	0.0	0.851064
28537	1.0	1.0	0.333333	0.0	0.893617

```
X = np.array(np.array(df.iloc[:,0:4].values),dtype=int)
X
```

```
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 0, 0, 0],
       ...,
       [1, 1, 1, 0],
       [1, 0, 0, 1],
       [1, 1, 1, 0]])
```

```
y = np.array(df.iloc[:,4],dtype=int)[np.newaxis].T
```

Step 8: Provide inputs to Multiple Back Propagation neural networks and predict patterns (Input Layer = 4, Hidden Layer = 7 and Output Layer = 1)

```
class Neural_Network(object):
  def __init__(self):
    #parameters
    self.inputSize = 4
    self.outputSize = 1
    self.hiddenSize = 7

    #weights
    self.W1 = np.random.randn(self.inputSize, self.hiddenSize) # (3x2) weight matrix from input to hidden layer
    self.W2 = np.random.randn(self.hiddenSize, self.outputSize) # (3x1) weight matrix from hidden to output layer

  def forward(self, X):
    #forward propagation through our network
    self.z = np.dot(X, self.W1) # dot product of X (input) and first set of 3x2 weights
    self.z2 = self.sigmoid(self.z) # activation function
    self.z3 = np.dot(self.z2, self.W2) # dot product of hidden layer (z2) and second set of 3x1 weights
    o = self.sigmoid(self.z3) # final activation function
    return o
```

Step 9: Use sigmoid function to activate the neurons

```
def sigmoid(self, s):
  # activation function
  return 1/(1+np.exp(-s))

def sigmoidPrime(self, s):
  #derivative of sigmoid
  return s * (1 - s)

def backward(self, X, y, o):
  # backward propgate through the network
  self.o_error = y - o # error in output
  self.o_delta = self.o_error*self.sigmoidPrime(o) # applying derivative of sigmoid to error

  self.z2_error = self.o_delta.dot(self.W2.T) # z2 error: how much our hidden layer weights contributed to output error
  self.z2_delta = self.z2_error*self.sigmoidPrime(self.z2) # applying derivative of sigmoid to z2 error

  self.W1 += X.T.dot(self.z2_delta) # adjusting first set (input --> hidden) weights
  self.W2 += self.z2.T.dot(self.o_delta) # adjusting second set (hidden --> output) weights

def train (self, X, y):
  o = self.forward(X)
  self.backward(X, y, o)
```

Step 10: Predict Pattern id and Loss on Test Data

```
NN = Neural_Network()
for i in range(100): # trains the NN 1,000 times
  print ("Input: \n" + str(X) )
  print ("Actual Output: \n" + str(y) )
  print ("Predicted Output: \n" + str(NN.forward(X)) )
  print ("Loss: \n" + str(np.mean(np.square(y - NN.forward(X))))) # mean sum squared loss
  print ("\n")
  NN.train(X, y)
```

```
Loss:
0.126
```

The results of the tests with the neural networks shows how it can be trained to recognize groups of players by their characteristics and learn the unique patterns of Guild, Level, Race, Class combinations of these groups in the game.

2. Time series forecasting to predict number of players

Another algorithm applied so far on this dataset is ARIMA model **Auto-Regressive Integrated Moving Averages** to predict number of players expected on a future time series

Step1: Load and handle Time series data in Python

```
df1.head()
```

	QueryTime	AvatarID
0	12/31/05 23:59:46	0
1	12/31/05 23:59:46	1
2	12/31/05 23:59:52	2
3	12/31/05 23:59:52	3
4	12/31/05 23:59:52	4

```
df2 = df1.groupby(['QueryTime']).size().reset_index()
```

```
df2.rename(columns = {0: 'PlayersCount'},  inplace=True)
```

```
dateparse = lambda dates: pd.to_datetime(dates, dayfirst=True)
```

```
#Strip until date with .str[:-9]
df2['Date'] = df2['QueryTime'].str[:-9]
```

```
df2 = df2[['Date','PlayersCount']]
```

```
df3 = df2.groupby(['Date']).sum().reset_index()
```

```
dateparse = lambda dates: pd.to_datetime(dates, dayfirst=True)
```

```
df3['DateTime'] = df3.apply(lambda col: dateparse(col['Date']), axis = 1)
```

```
data = df3[['DateTime', 'PlayersCount']].set_index(['DateTime'])
```

```
data.index
```

```
DatetimeIndex(['2005-12-31', '2006-01-01', '2006-01-02', '2006-01-03',
               '2006-01-04', '2006-01-05', '2006-01-06', '2006-01-07',
               '2006-01-08', '2006-01-09',
               ...
               '2009-01-01', '2009-02-01', '2009-03-01', '2009-04-01',
               '2009-05-01', '2009-06-01', '2009-07-01', '2009-08-01',
               '2009-09-01', '2009-10-01'],
              dtype='datetime64[ns]', name=u'DateTime', length=1083, freq=None)
```

```
data.sort_index(inplace = True)
```

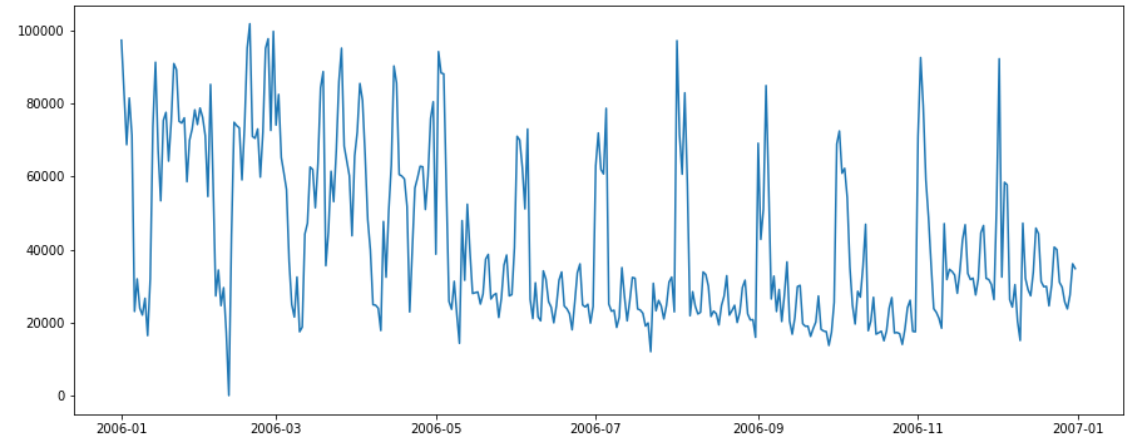
```
ts = data['PlayersCount']
ts.head(10)
```

```
DateTime
2005-12-31       26
2006-01-01    97322
2006-01-02    82668
2006-01-03    68720
2006-01-04    81508
2006-01-05    71252
2006-01-06    23068
2006-01-07    31981
2006-01-08    23990
2006-01-09    22086
Name: PlayersCount, dtype: int64
```

Step 2: Check stationarity of a Time series

```
plt.plot(ts['2006'])
```

```
[<matplotlib.lines.Line2D at 0x510cdb70>]
```



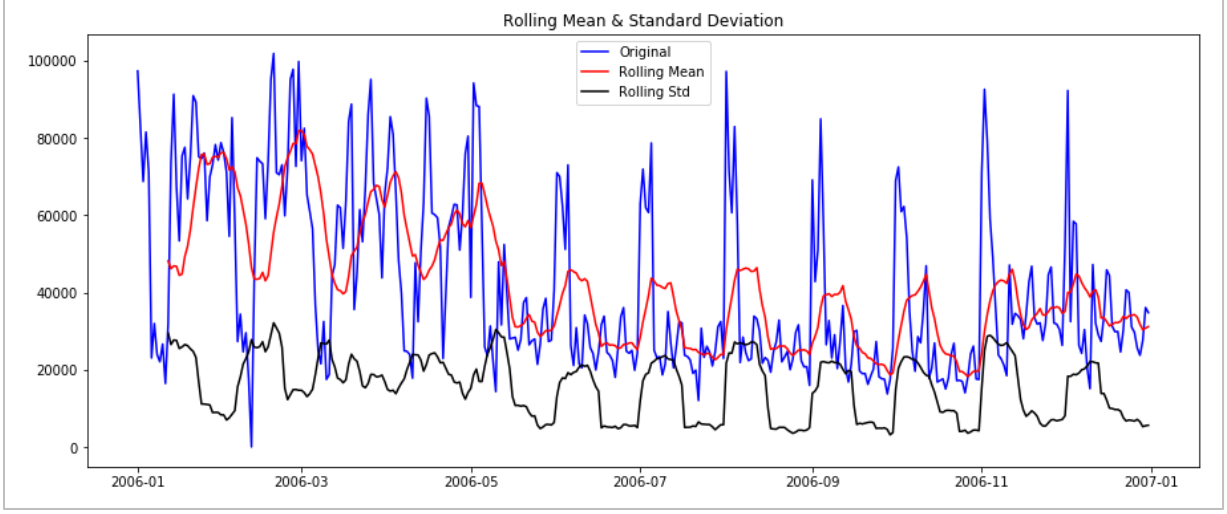

```
from statsmodels.tsa.stattools import adfuller
def test_stationarity(timeseries):

    #Determing rolling statistics
    rolmean = timeseries.rolling(window=12,center=False).mean()
    rolstd = timeseries.rolling(window=12,center=False).std()

    #Plot rolling statistics:
    orig = plt.plot(timeseries, color='blue',label='Original')
    mean = plt.plot(rolmean, color='red', label='Rolling Mean')
    std = plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show(block=False)

    #Perform Dickey-Fuller test:
    print ('Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used','Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print (dfoutput)
```

```
test_stationarity(ts['2006'])
```



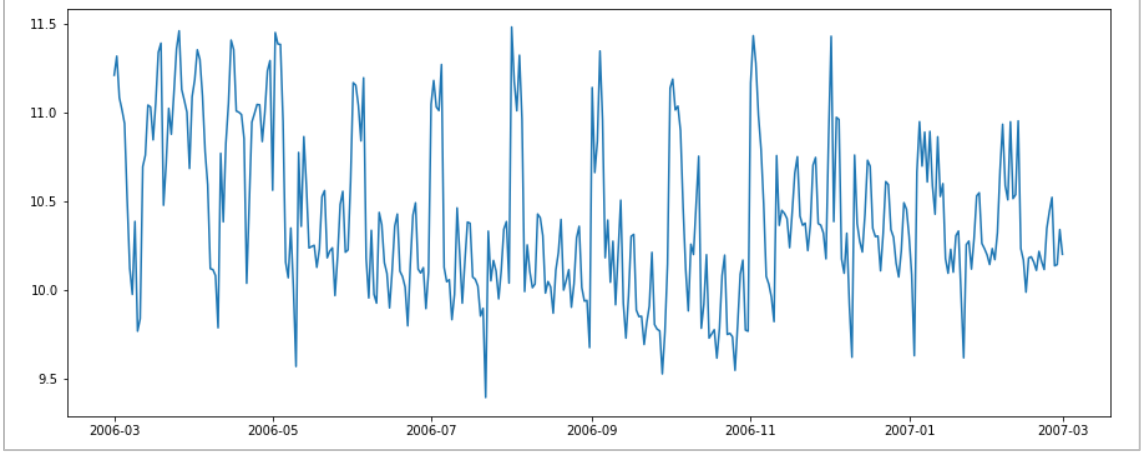

```
Results of Dickey-Fuller Test:
Test Statistic                  -4.586999
p-value                          0.000136
#Lags Used                       6.000000
Number of Observations Used    357.000000
Critical Value (1%)             -3.448801
Critical Value (5%)             -2.869670
Critical Value (10%)            -2.571101
dtype: float64
```

Step 3: Make time series stationary

```
Estimating & Eliminating Trend
```

```
#ts_log = np.log(ts['2006'])
ts_log = np.log(ts['2006-03-01':'2007-03-01'])
plt.plot(ts_log)
```

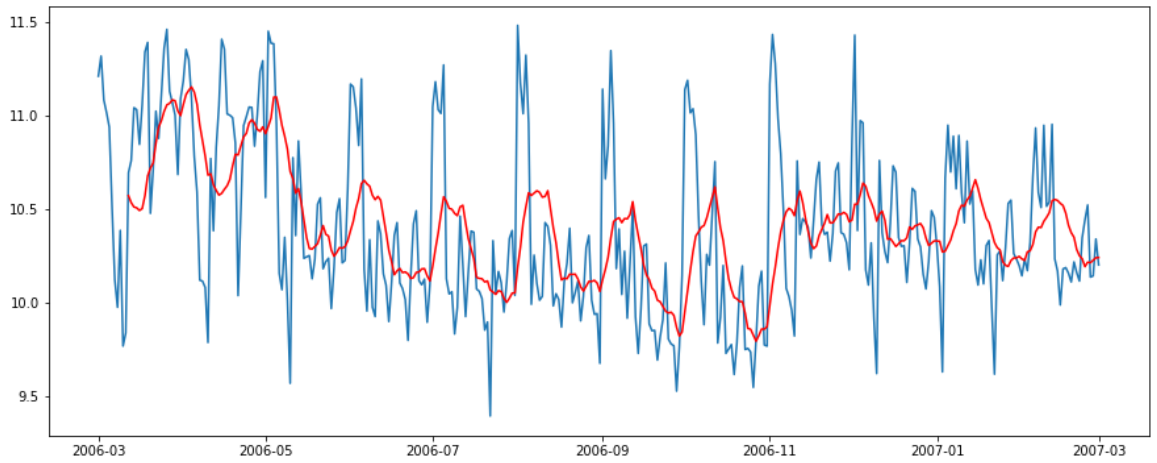
```
[<matplotlib.lines.Line2D at 0x5228e908>]
```



Moving average

```
moving_avg = ts_log.rolling(window=12,center=False).mean()
plt.plot(ts_log)
plt.plot(moving_avg, color='red')
```

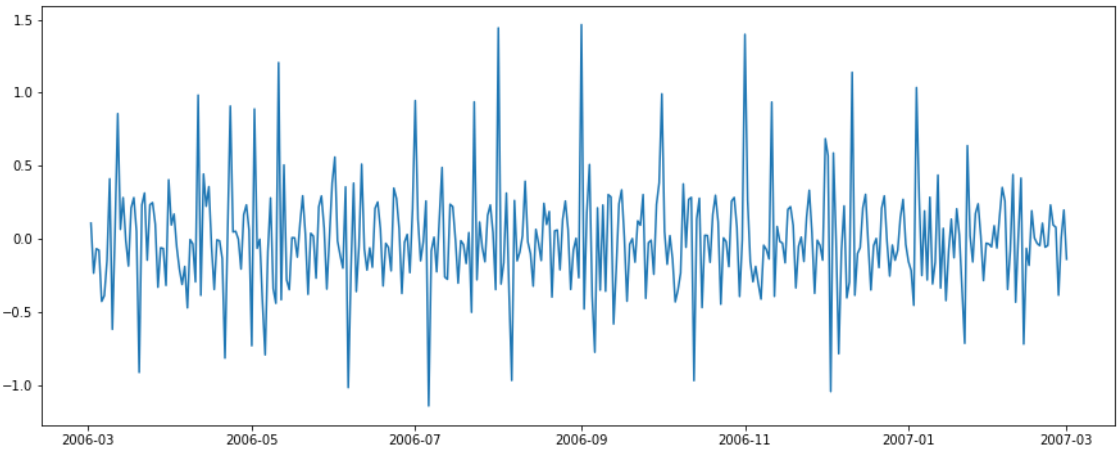
```
[<matplotlib.lines.Line2D at 0x52852550>]
```



Eliminating Trend and Seasonality

Differencing – taking the differece with a particular time lag

```
ts_log_diff = ts_log - ts_log.shift()
plt.plot(ts_log_diff)
```

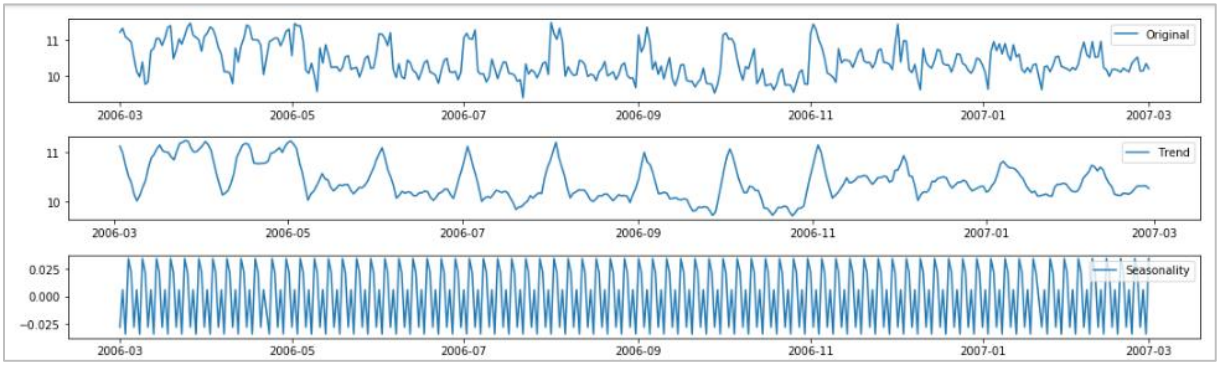


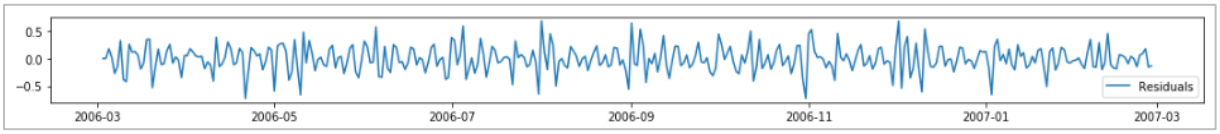
Decomposition – modeling both trend and seasonality and removing them from the model

```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(ts_log, freq = 5)

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.subplot(411)
plt.plot(ts_log, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```



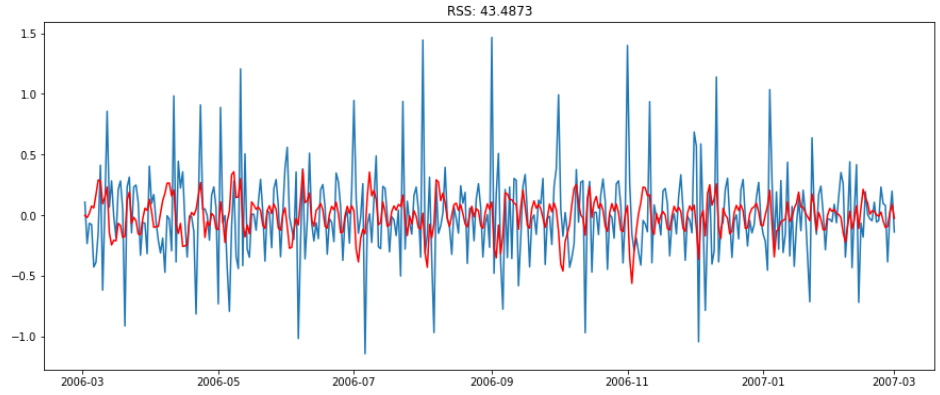


Step 4: Forecast on Time series

Applying ARIMA model

```
model = ARIMA(ts_log, order=(0, 1, 2))
results_ARIMA = model.fit(disp=-1)
plt.plot(ts_log_diff)
plt.plot(results_ARIMA.fittedvalues, color='red')
plt.title('RSS: %.4f'% sum((results_ARIMA.fittedvalues-ts_log_diff)**2))
```

```
Text(0.5,1,'RSS: 43.4873')
```




```
predictions_ARIMA_diff = pd.Series(results_ARIMA.fittedvalues, copy=True)
print (predictions_ARIMA_diff.head())
```

```
DateTime
2006-03-02   -0.002225
2006-03-03   -0.019941
2006-03-04    0.014380
2006-03-05    0.074339
2006-03-06    0.060022
dtype: float64
```

```
predictions_ARIMA_diff_cumsum = predictions_ARIMA_diff.cumsum()
print (predictions_ARIMA_diff_cumsum.head())
```

```
DateTime
2006-03-02   -0.002225
2006-03-03   -0.022166
2006-03-04   -0.007786
2006-03-05    0.066553
2006-03-06    0.126575
dtype: float64
```

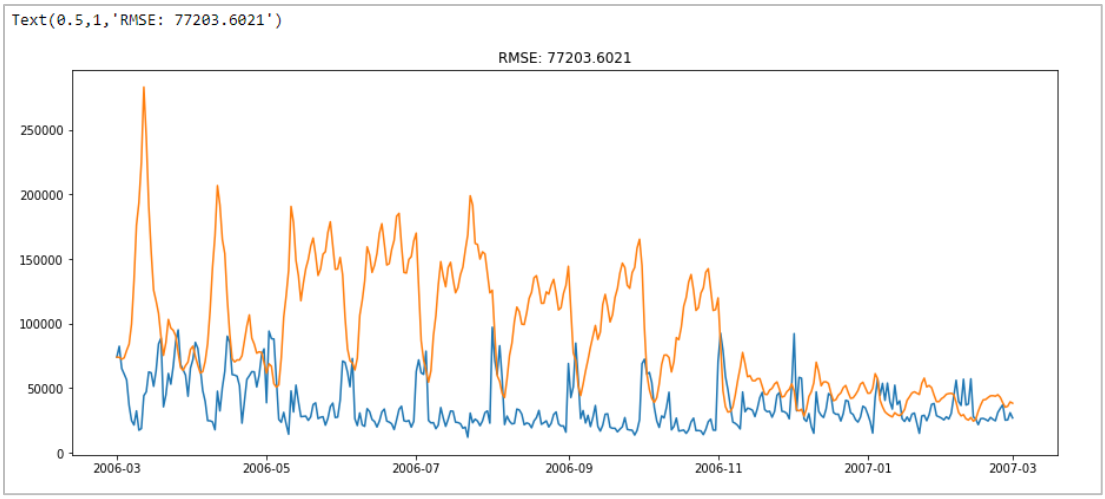
```
predictions_ARIMA_log = pd.Series(ts_log.ix[0], index=ts_log.index)
predictions_ARIMA_log = predictions_ARIMA_log.add(predictions_ARIMA_diff_cumsum,fill_value=0)
predictions_ARIMA_log.head()
```

```
DateTime
2006-03-01    11.213063
2006-03-02    11.210838
2006-03-03    11.190897
2006-03-04    11.205277
2006-03-05    11.279616
dtype: float64
```

```
predictions_ARIMA = np.exp(predictions_ARIMA_log)
plt.plot(ts['2006-03-01':'2007-03-01'])
plt.plot(predictions_ARIMA)
plt.title('RMSE: %.4f'% np.sqrt(sum((predictions_ARIMA-ts['2006-03-01':'2007-03-01'])**2)/len(ts['2006-03-01':'2007-03-01'])))
```

```
Text(0.5,1,'RMSE: 77203.6021')
```

```
Text(0.5,1,'RMSE: 77203.6021')
```



References

i. **Big Data Analytics Using Neural networks**
***Author**: Chetan Sharma*
***Master's Thesis:** San Jose State University*

ii. **Game Analytics - Maximizing the Value of Player Data**
***Authors:** Magy Seif El-Nasr, Anders Drachen and Alessandro Canossa*
***Publisher:** Springer*

iii. **Big Data Analytics in Cloud Gaming**
***Authors:** Victor Perazzolo Barros and Pollyana Notargiacomo*
Paper: 2016 IEEE International Conference on Big Data

iv. **Setting Players' Behaviors in World of Warcraft through Semi-Supervised Learning**
***Authors:** Marcelo Souza Nery, Victor do Nascimento Silva, Roque Anderson S. Teixeira and Adriano Alonso Veloso*