

# Data Exploration Technical Appendix

Elkin Andres Villa Sanchez

# Data Set Overview

The table below lists each of the files available for analysis with a short description of what is found in each one.

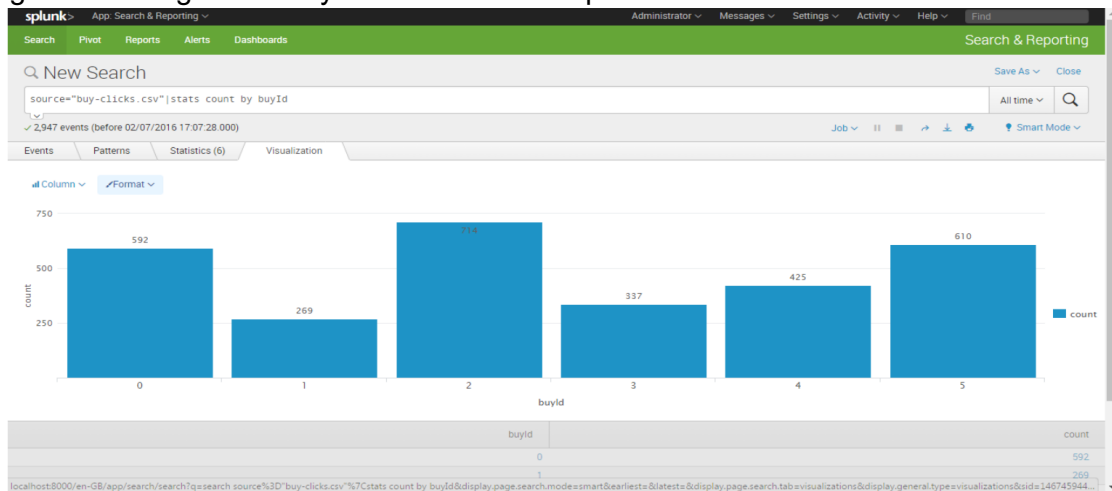
| File Name      | Description  | Fields  |
|----------------|--|---|
| ad-clicks.csv  | A line is added to this file when a player clicks on an advertisement in the Flamingo app. | <ul style="list-style-type: none"><li>• timestamp: when the click occurred.</li><li>• txId: a unique id (within ad-clicks.log) for the click</li><li>• userSessionid: the id of the user session for the user who made the click</li><li>• teamid: the current team id of the user who made the click</li><li>• userid: the user id of the user who made the click</li><li>• adId: the id of the ad clicked on</li><li>• adCategory: the category/type of ad clicked on</li></ul>         |
| buy-clicks.csv | A line is added to this file when a player makes an in-app purchase in the Flamingo app.   | <ul style="list-style-type: none"><li>• timestamp: when the purchase was made.</li><li>• txId: a unique id (within buy-clicks.log) for the purchase</li><li>• userSessionId: the id of the user session for the user who made the purchase</li><li>• team: the current team id of the user who made the purchase</li><li>• userId: the user id of the user who made the purchase</li><li>• buyId: the id of the item purchased</li><li>• price: the price of the item purchased</li></ul> |
| users.csv      | This file contains a line for each user playing the game.                                  | <ul style="list-style-type: none"><li>• timestamp: when user first played the game.</li><li>• userId: the user id assigned to the user.</li><li>• nick: the nickname chosen by the user.</li><li>• twitter: the twitter handle of the user.</li><li>• dob: the date of birth of the user.</li><li>• country: the two-letter country code where the user lives.</li></ul>  |
| team.csv       | This file contains a line for each team terminated in the game.                            | <ul style="list-style-type: none"><li>• teamId: the id of the team</li><li>• name: the name of the team</li><li>• teamCreationTime: the timestamp when the team was created</li><li>• teamEndTime: the timestamp when the last member left the team</li></ul>   |

|                      |   |   |
|----------------------|---|---|
|                      |   | <ul style="list-style-type: none"> <li>• strength: a measure of team strength, roughly corresponding to the success of a team</li> <li>• currentLevel: the current level of the team</li> </ul>   |
| team-assignments.csv | A line is added to this file each time a user joins a team. A user can be in at most a single team at a time.   | <ul style="list-style-type: none"> <li>• timestamp: when the user joined the team.</li> <li>• team: the id of the team</li> <li>• userId: the id of the user</li> <li>• assignmentId: a unique id for this assignment</li> </ul>  |
| level-events.csv     | A line is added to this file each time a team starts or finishes a level in the game.   | <ul style="list-style-type: none"> <li>• timestamp: when the event occurred.</li> <li>• eventId: a unique id for the event</li> <li>• teamId: the id of the team</li> <li>• teamLevel: the level started or completed</li> <li>• eventType: the type of event, either start or end</li> </ul>   |
| user-session.csv     | Each line in this file describes a user session, which denotes when a user starts and stops playing the game. Additionally, when a team goes to the next level in the game, the session is ended for each user in the team and a new one started. | <ul style="list-style-type: none"> <li>• timestamp: a timestamp denoting when the event occurred.</li> <li>• userSessionId: a unique id for the session.</li> <li>• userId: the current user's ID.</li> <li>• teamId: the current user's team.</li> <li>• assignmentId: the team assignment id for the user to the team.</li> <li>• sessionType: whether the event is the start or end of a session.</li> <li>• teamLevel: the level of the team during this session.</li> <li>• platformType: the type of platform of the user during this session.</li> </ul> |
| game-clicks.csv      | A line is added to this file each time a user performs a click in the game.   | <ul style="list-style-type: none"> <li>• timestamp: when the click occurred.</li> <li>• clickId: a unique id for the click.</li> <li>• userId: the id of the user performing the click.</li> <li>• userSessionId: the id of the session of the user when the click is performed.</li> <li>• isHit: denotes if the click was on a flamingo (value is 1) or missed the flamingo (value is 0)</li> <li>• teamId: the id of the team of the user</li> <li>• teamLevel: the current level of the team of the user</li> </ul>   |

## 1.1 Aggregation

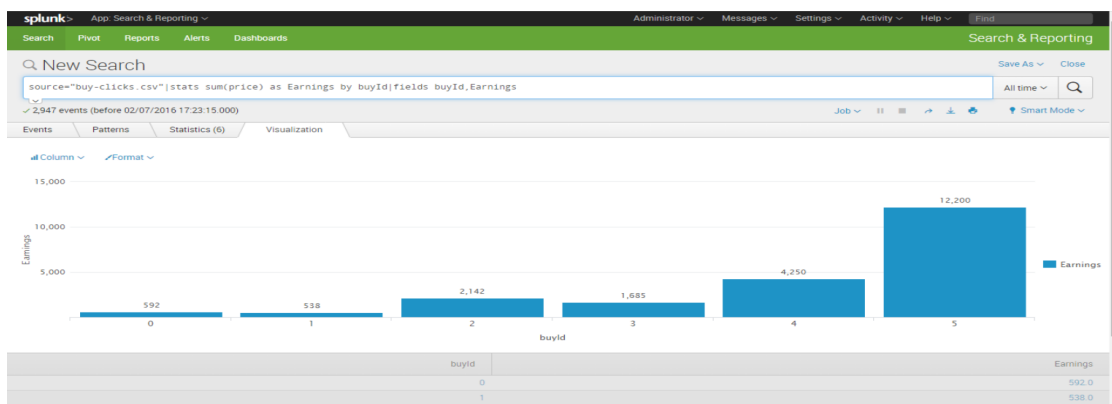
|  |       |
|--|-------|
| Amount spent buying items                | 21407 |
| # Unique items available to be purchased | 6     |

A histogram showing how many times each item is purchased:



Using the “buy-clicks.csv” file, we can make the histogram above, it shows the times that each item is purchased. Among six items, the item “2” is the most purchased, the item “1” is the least purchased.

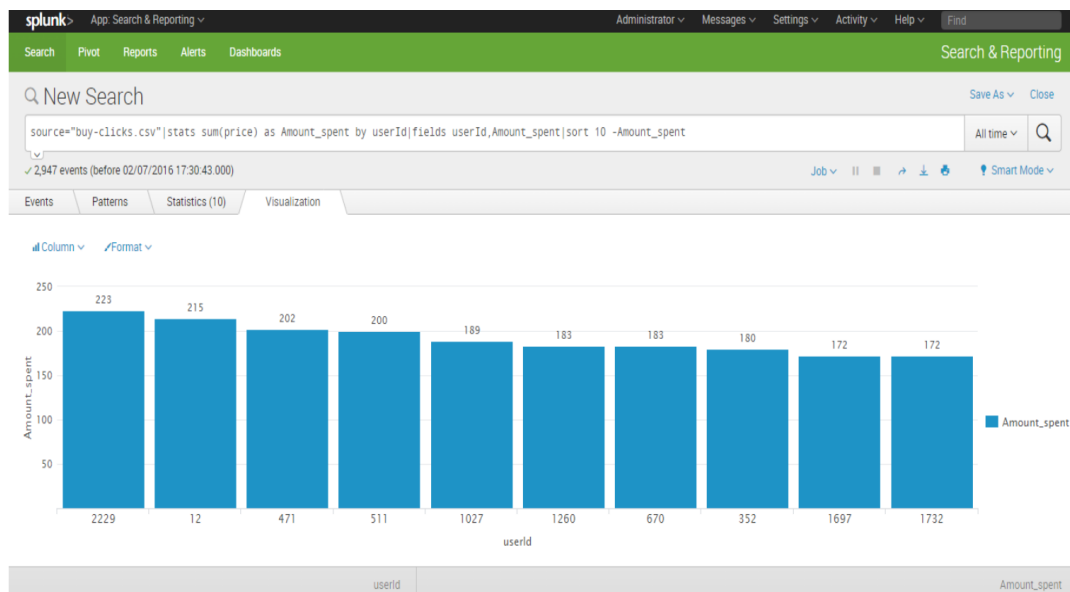
A histogram showing how much money was made from each item:



Making this histogram with the “buy-clicks.csv” file, we get different amount of money that made from each item. Among them, the item “5” made the most money, and the item “1” made the least money, which is also the least purchased, it means the item “1” is not preferred by most people. In this case, providers should think about strategies to change this situation or to solve this problem.

## 1.2 Filtering

A histogram showing total amount of money spent by the top ten users (ranked by how much money they spent).



Thanks to the “buy-clicks.csv” file, the histogram above could be made, it shows the top ten users according to their total amount of spending. The user whose userId is “2229” spends the most, his spending is nearly 225 units.

The following table shows the user id, platform, and hit-ratio percentage for the top three buying users:

| Rank | User Id | Platform | Hit-Ratio (%) |
|------|---------|----------|---------------|
| 1    | 2229    | iphone   | 11.60         |
| 2    | 12      | iphone   | 13.07         |
| 3    | 471     | iphone   | 14.50         |

According to the histogram above, we know the userId of top three users are “2229”, “12” and “471”. In order to check their platform, we can use the file “user-session.csv”. Then with the file “game-clicks.csv”, we can calculate the Hit-Ratio by  $\text{sum(isHit)/count(isHit)}$  for each user.

## Part 2 KNIME Classification Analysis

## 2.1 Data Preparation

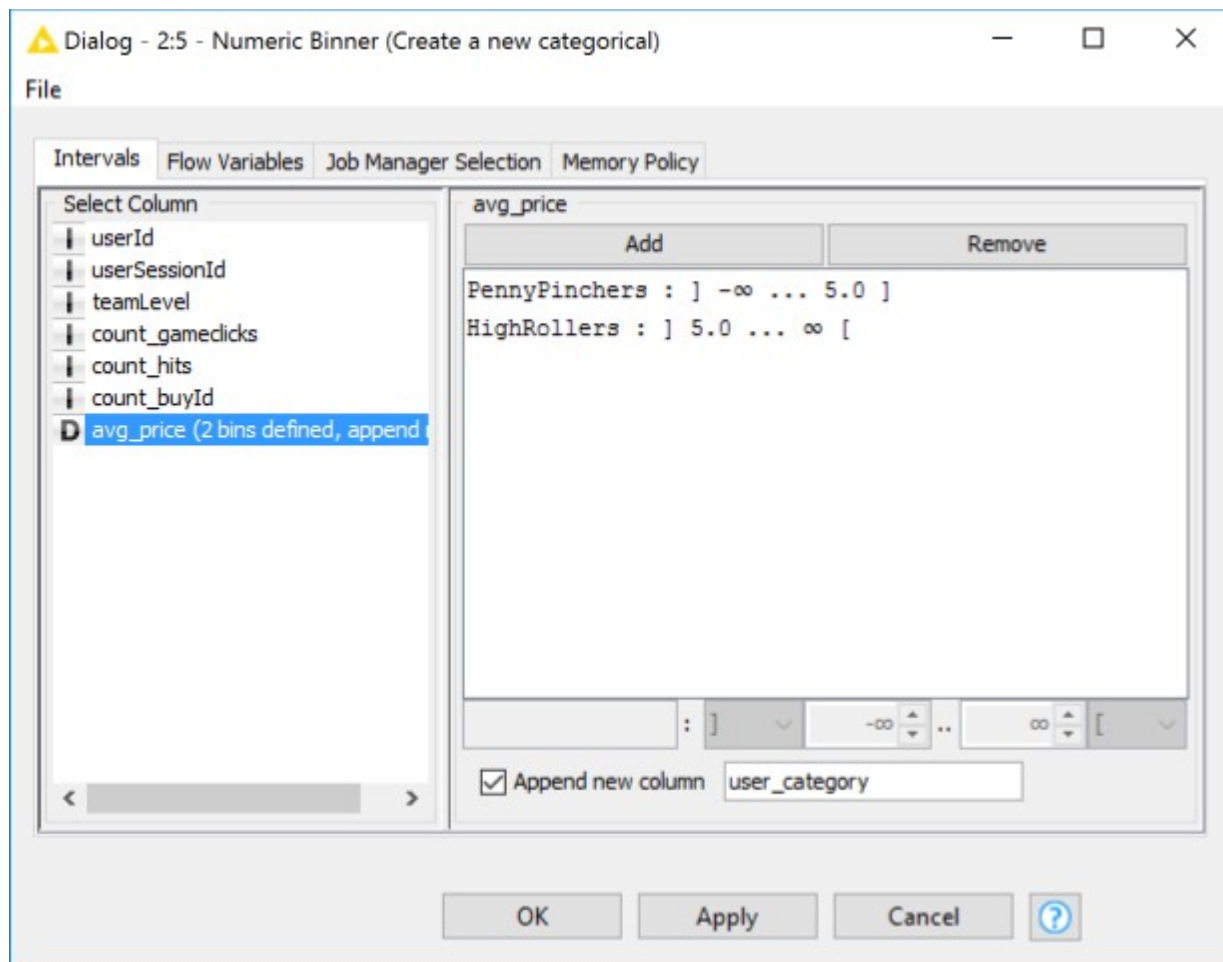
Analysis of combined\_data.csv to investigate the classification of users.

### 2.1.1 Sample Selection

| Item                        | Amount |
|-----------------------------|--------|
| # of Samples                | 4619   |
| # of Samples with Purchases | 1411   |

### 2.1.2 Attribute Creation

A new categorial attribute was created to enable analysis of players as broken into 2 categories (HighRollers and PennyPinchers). A screenshot of the attribute follows:



A new categorical attribute, named “user\_category”, is created by the Numeric Binner node. As presented in the instruction, we need to define two categories for price which we will use to distinguish between HighRollers(buyers of items that cost more than \$5.00) and PennyPinchers (buyers of items that cost \$5.00 or less), so as we see in the screenshot above, the user who costs \$5.00 or less is defined as “PennyPinchers”, the user who costs more than \$5.00 is defined as “HighRollers”.

The creation of this new categorical attribute was necessary because it can facilitate the classification of users and contribute to the following steps.

### 2.1.3 Attribute Selection

The following attributes were filtered from the dataset for the following reasons:

| Attribute     | Rationale for Filtering  |
|---------------|--|
| userId        | Since the objective is to predict which user is likely to purchase big-ticket items, and the attribute “userId” has no effect on it, so it’s removed.        |
| userSessionId | Since the objective is to predict which user is likely to purchase big-ticket items, and the attribute “userSessionId” has no effect on it, so it’s removed. |
| avg_price     | Since a new attribute “user_category” has been created, which was generated from the attribute “avg_price”, so we can remove it.                             |



# Data Partitioning and Modeling

The data was partitioned into train and test datasets.

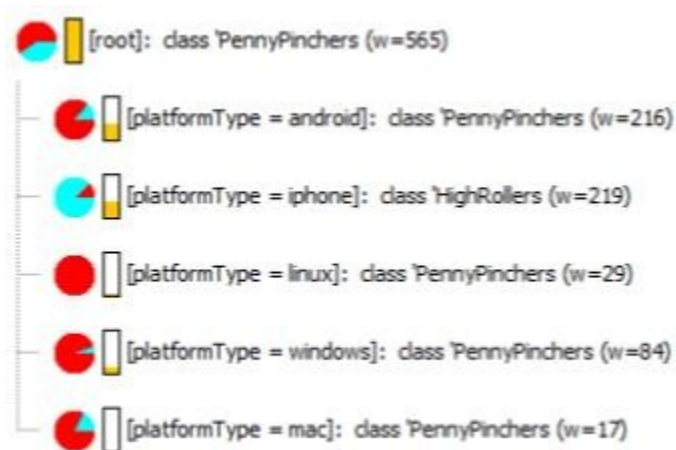
The training data set was used to create the decision tree model.

The trained model was then applied to the test dataset.

This is important because train data set is used in creating the decision tree model, the apply the model to the test data set, which is not used to train the mode then we can see the accuracy of the model.

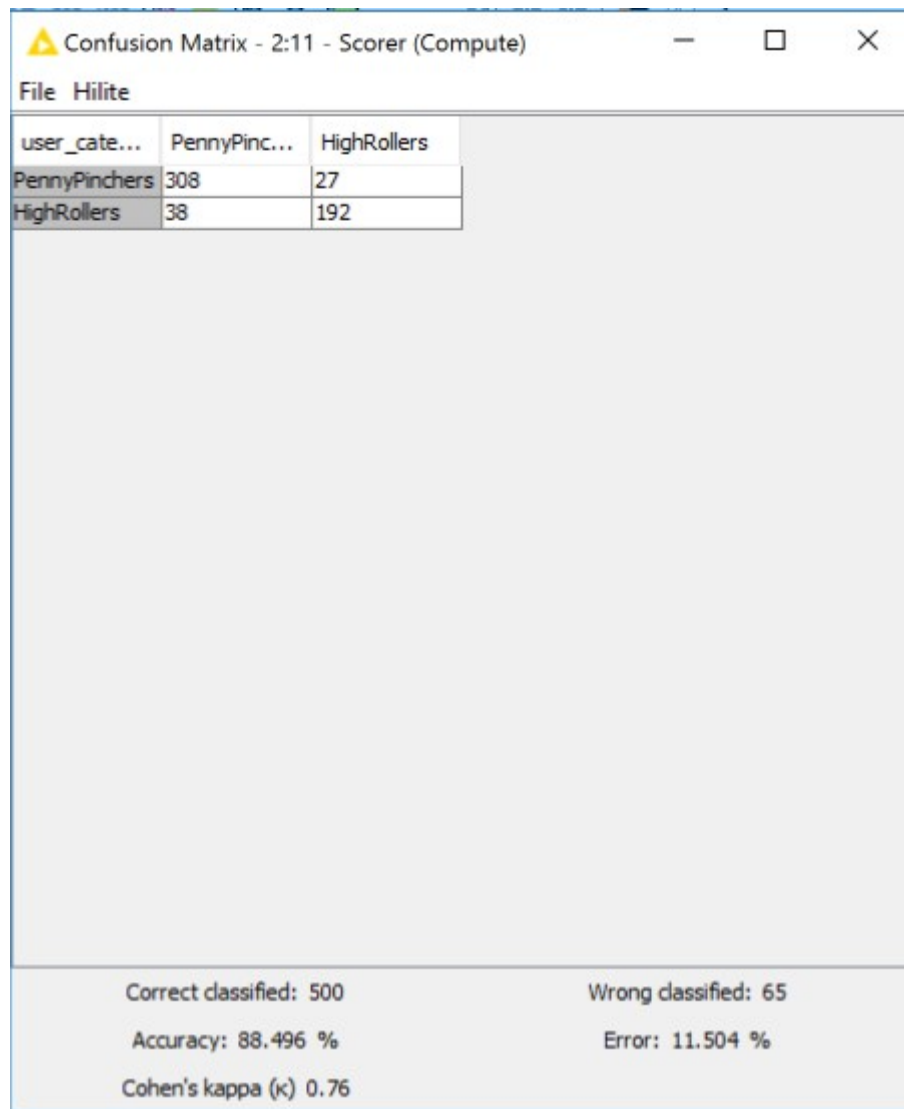
When partitioning the data using sampling, it is important to set the random seed because it can get the same data partitions every time the node is executed.

A screenshot of the resulting decision tree can be seen below:



# Evaluation

A screenshot of the confusion matrix can be seen below:



The screenshot shows a window titled "Confusion Matrix - 2:11 - Scorer (Compute)". Inside, there is a table with the following data:

| user_cate...  | PennyPinc... | HighRollers |
|---------------|--------------|-------------|
| PennyPinchers | 308          | 27          |
| HighRollers   | 38           | 192         |

Below the table, the following statistics are displayed:

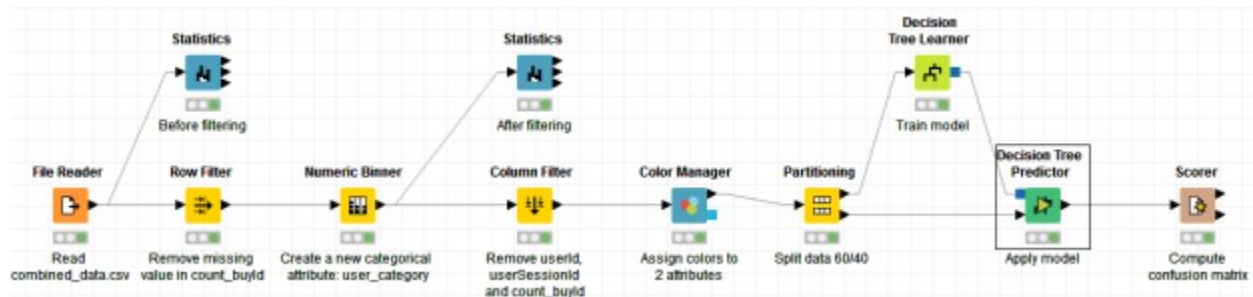
- Correct classified: 500
- Wrong classified: 65
- Accuracy: 88.496 %
- Error: 11.504 %
- Cohen's kappa ( $\kappa$ ) 0.76

As seen in the screenshot above, the overall accuracy of the model is 88.496%.

- “308” & “38”: according to the model, we predict that 348(308+38) users are PennyPinchers, but among them 308 users are truly predicted, which means among these 348 users, 308 users are exactly PennyPinchers, 38 HighRollers are incorrectly predicted as PennyPinchers.
- “192” & “27”: according to the model, we predict that 219(192+27) users are HighRollers, but among them 192 users are truly predicted, which means among these 219 users, 192 users are exactly HighRollers, 27 PennyPinchers are incorrectly predicted as HighRollers.

# Analysis Conclusions

The final KNIME workflow is shown below:



According to the resulting decision tree, it obviously shows that the predicted user\_category is different in various platforms, the users on the platform android, linux, windows and mac are almost PennyPincher, however, most users which on the platform iphone are HighRoller.

| Specific Recommendations to Increase Revenue                                |
|---|
| 1. Offer more products to iPhone users.                                     |
| 2. Offer some promotions to PennyPinchers for attracting their consumption. |

## Part 3 Spark MLlib Clustering

## Attribute Selection

| Attribute                        | Rationale for Selection  |
|----------------------------------|--|
| amount of ad-clicking per user   | according to this attribute, we can capture users' behavior on clicking ad   |
| amount of game-clicking per user | according to this attribute, we can capture users' behavior on clicking game |
| total price spent by each user   | total cost of each user can capture preference degree of each user           |

## Training Data Set Creation

The training data set used for this analysis is shown below (first 5 lines):

### Create the final training dataset

Our training data set is almost ready. At this stage we can remove the 'userid' from each row, since 'userid' is a computer generated random number assigned to each user. It does not capture any behavioral aspect of a user. One way to drop the 'userid', is to select the other two columns.

```
In [37]: training_df = combined_df[['totalAdClicks', 'totalGameClicks', 'revenue']]
training_df.head(5)
```

```
Out[37]:
```

|   | totalAdClicks | totalGameClicks | revenue |
|---|---------------|-----------------|---------|
| 0 | 44            | 716             | 21.0    |
| 1 | 10            | 380             | 53.0    |
| 2 | 37            | 508             | 80.0    |
| 3 | 19            | 3107            | 11.0    |
| 4 | 46            | 704             | 215.0   |

### Display the dimensions of the training dataset

Display the dimension of the training data set. To display the dimensions of the training\_df, simply add .shape as a suffix and hit enter.

```
In [38]: training_df.shape
```

```
Out[38]: (543, 3)
```

Dimensions of the training data set (rows x columns) : 543 \* 3

# of clusters created: 3

## Cluster Centers

| Cluster # | Cluster Center                                      |
|-----------|---|
| 1         | array( [ 25.12037037, 362.50308642, 35.35802469 ] ) |
| 2         | array( [ 32.05, 2393.95, 41.2 ] )                   |
| 3         | array( [ 36.47486034, 953.82122905, 46.16201117 ] ) |

These clusters can be differentiated from each other as follows:

The first number (field 1) in each array refers to the number of ads per user click, the second number (field 2) in each array refers to amount of game-clicking per user and the third number (field 3) is the cost on this game of each user.

Cluster 1 is different from the others in that the users' ad-clicks, game-clicks and cost are all less than others, this kind of users can be called "low level spending user".

Cluster 3 is different from the others in that the users' ad-clicks, game-clicks and cost are all more than others, this kind of users can be called "high level spending user".

Cluster 2 is different from the others in that the ad-clicks is not the least, game-clicks is the most but their cost is not the most, this kind of users can be called "neutral user".



## Neo4j Chat Graph Analysis

# Modeling Chat Data using a Graph Data Model

Using a Graph Data Model to illustrate the chatting interaction among users with Chat Data. A user in a team can create a chat session and then create chat (i.e. chat item) in the chat session. Otherwise, a user could be mentioned by a chat item, and a chat item can response to another chat item, which represent the communication among the users in the same team. Moreover, a user can also join in an existed team chat session or leave it.

## Creation of the Graph Database for Chats

### CSV Files

| File Name                  | Description       | Fields  |
|----------------------------|-------------------|---|
| chat_create_team_chat.csv  | userid            | the user id assigned to the user                                      |
|                            | teamid            | the id of the team  |
|                            | TeamChatSessionID | a unique id for the chat session                                      |
|                            | timestamp         | a timestamp denoting when the chat session created                    |
| chat_item_team_chat.csv    | userid            | the user id assigned to the user                                      |
|                            | teamchatsessionid | a unique id for the chat session                                      |
|                            | chatitemid        | a unique id for the chat item   |
|                            | timestamp         | a timestamp denoting when the chat item created                       |
| chat_join_team_chat.csv    | userid            | the user id assigned to the user                                      |
|                            | TeamChatSessionID | a unique id for the chat session                                      |
|                            | timestamp         | a timestamp denoting when the user join in a chat session             |
| chat_leave_team_chat.csv   | userid            | the user id assigned to the user                                      |
|                            | teamchatsessionid | a unique id for the chat session                                      |
|                            | timestamp         | a timestamp denoting when the user leave a chat session               |
| chat_mention_team_chat.csv | ChatItemId        | the id of the ChatItem  |
|                            | userid            | the user id assigned to the user                                      |
|                            | timeStamp         | a timestamp denoting when the user mentioned by a chat item           |
| chat_respond_team_chat.csv | chatid1           | the id of the chat post 1   |
|                            | chatid2           | the id of the chat post 2   |
|                            | timestamp         | a timestamp denoting when the chat post 1 responds to the chat post 2 |



## Loading Process

Using Cypher Query Language to load the CSV data into neo4j, each row of script is parsed for refine the nodes, the edges and its timestamp. Let's consult the following script as an example:

```
LOAD CSV FROM "file:///chat-data/chat_item_team_chat.csv" AS row
MERGE (u:User {id: toInt(row[0])})
MERGE (c:TeamChatSession {id: toInt(row[1])})
MERGE (i:ChatItem {id: toInt(row[2])})
MERGE (u)-[:CreateChat{timeStamp: row[3]}]->(i)
MERGE (i)-[:PartOf{timeStamp: row[3]}]->(c)
```

The first line gives the path of the file, this command reads the chat\_item\_team\_chat.csv at a time and create user nodes. The 0<sup>th</sup> column value is converted to an integer and is used to populate the id attribute. Similarly the other nodes are created.

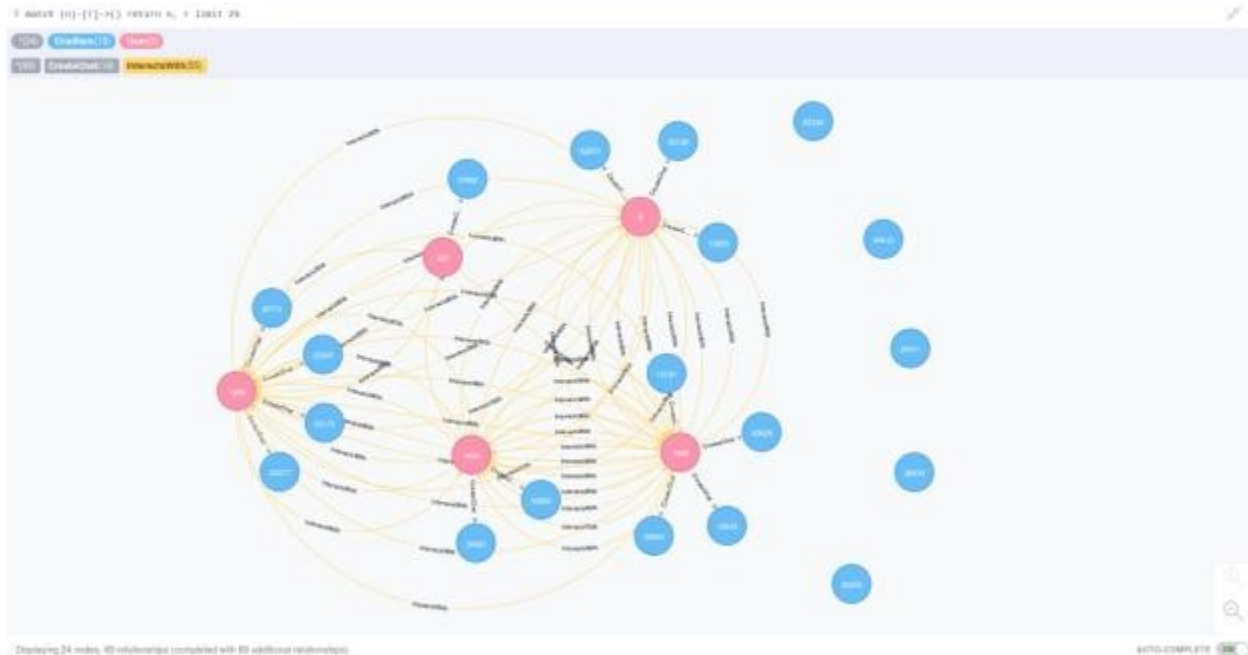
Line 5, MERGE (u)-[:CreateChat{timeStamp: row[3]}]->(i) creates an edge labeled "CreateChat" between the User node u and the ChatItem node i. This edge has a property called timeStamp. This property is filled by the content of column 3 of the same row.

Line 6, MERGE (i)-[:PartOf{timeStamp: row[3]}]->(c) creates an edge labeled "PartOf" between the ChatItem node i and the TeamChatSession node c. This edge has a property called timeStamp. This property is filled by the content of column 3 of the same row.

## A Screenshot of Some Part of the Graph

The graphs must include clearly visible examples of most node and edge types. Below are two acceptable examples. The first example is a rendered in the default Neo4j distribution, the second has had some nodes moved to expose the edges more clearly. Both include examples of most node and edge types.





## The longest conversation chain and its participants

### ➤ Query

```
match p = (i1)-[:ResponseTo*]->(i2)
return length(p)
order by length(p) desc limit 1
```

### ➤ Screenshot



- how many unique users were part of this chain?  
Since we've already known the longest conversation chain has path length of 9, next we will find all the distinct users which are part of this path. After running the following query, **5 unique users** are extracted.

➤ Query

```
match p = (i1)-[:ResponseTo*]->(i2)
where length(p) = 9
with p
match (u)-[:CreateChat]->(i)
where i in nodes(p)
return count(distinct u)
```

➤ Screenshot

The screenshot shows a query execution interface. At the top, the query is displayed: `match p = (i1)-[:ResponseTo*]->(i2) where length(p)=9 with p match (u)-[:CreateChat]->(i) where i in nodes(p) return count(distinct u)`. Below the query, the result is shown in a table with one column, `count(distinct u)`, and one row with the value `5`. The interface includes a sidebar with icons for a table, text, and code. At the bottom, a status message reads: "Started streaming 1 record after 194 ms and completed after 194 ms."

| count(distinct u) |
|-------------------|
| 5                 |

# The relationship between top 10 chattiest users and top 10 chattiest teams

Describe your steps from Question 2. In the process, create the following two tables. You only need to include the top 3 for each table. Identify and report whether any of the chattiest users were part of any of the chattiest teams.

## 4.1.1 Chattiest Users

We firstly match the CreateChat edge from User node to ChatItem node, then return the ChatItem amount per user, and order by the amount in descending order.

- Query

```
match (u)-[:CreateChat*]->(i)
return u.id, count(i)
order by count(i) desc limit 10
```

- Screenshot

The screenshot shows a query execution interface. At the top, the query is displayed: `5 match (u)-[:CreateChat*]->(i) return u.id, count(i) order by count(i) desc limit 10`. Below the query, there is a table with two columns: "u.id" and "count(i)". The table contains 10 rows of data, representing the top 10 chattiest users. The interface also includes a sidebar with icons for Rows, Text, and Code, and a status bar at the bottom indicating "Started streaming 10 records after 371 ms and completed after 371 ms."

| "u.id" | "count(i)" |
|--------|------------|
| "394"  | "115"      |
| "2067" | "111"      |
| "209"  | "109"      |
| "1087" | "109"      |
| "554"  | "107"      |
| "516"  | "105"      |
| "1627" | "105"      |
| "999"  | "105"      |
| "668"  | "104"      |
| "461"  | "104"      |

| Users             | Number of Chats |
|-------------------|-----------------|
| 394               | 115             |
| 2067              | 111             |
| 209               | 109             |
| 1087 <sub>1</sub> | 109             |

<sub>1</sub> Both user 209 and user 1087 create 109 chat items, we keep both.

#### 4.1.2 Chattiest Teams

We firstly match the `PartOf` edge from `ChatItem` node to `TeamChatSession` node, match the `OwnedBy` edge from `TeamChatSession` node to `Team` node, then return the `TeamChatSession` amount per team, and order by the amount in descending order.

- Query

```
match (i)-[:PartOf*]->(c)-[:OwnedBy*]->(t)
return t.id, count(c)
order by count(c) desc limit 10
```

- Screenshot

| t.id | count(c) |
|------|----------|
| 82   | 1324     |
| 185  | 1036     |
| 112  | 957      |
| 18   | 844      |
| 194  | 836      |
| 129  | 814      |
| 52   | 788      |
| 136  | 783      |
| 146  | 746      |
| 81   | 736      |

| Teams | Number of Chats |
|-------|-----------------|
| 82    | 1324            |
| 185   | 1036            |
| 112   | 957             |

## Final Result

Combine the two query above together as follows:

- Query

```
match (u)-[:CreateChat*]->(i)-[:PartOf*]->(c)-[:OwnedBy*]->(t)
return u.id, t.id, count(c)
order by count(c) desc limit 10
```

- Screenshot



The screenshot shows a database query result interface. At the top, the query is displayed: `match (u)-[:CreateChat*]->(i)-[:PartOf*]->(c)-[:OwnedBy*]->(t) return u.id, t.id, count(c) order by count(c) desc limit 10`. Below the query, there are icons for Rows, Text, and Code. The main area displays a table with 10 rows of results. The columns are labeled "u.id", "t.id", and "count(c)". The data is as follows:

| u.id | t.id | count(c) |
|------|------|----------|
| 394  | 63   | 115      |
| 2067 | 7    | 111      |
| 209  | 7    | 109      |
| 1087 | 77   | 109      |
| 554  | 181  | 107      |
| 1627 | 7    | 105      |
| 516  | 7    | 105      |
| 999  | 52   | 105      |
| 461  | 104  | 104      |
| 668  | 89   | 104      |

At the bottom of the interface, a status bar indicates: "Started streaming 10 records after 457 ms and completed after 457 ms." and "MAX COLUMN WIDTH: 100".

As result shows, the user 999, which in the team 52 is part of the top 10 chattiest teams, but other 9 users are not part of the top 10 chattiest teams. This states that most of the chattiest users are not in the chattiest teams.

## How Active Are Groups of Users?

- one Mentioned another user in a chat, one CreateChat in response to another user's ChatItem

```
match (u1:User)-[:CreateChat]->(i)-[:Mentioned]->(u2:User)
create (u1)-[:InteractsWith]->(u2)
```

- one created a ChatItem in response to another user's ChatItem

```
match (u1:User)-[:CreateChat]->(i1:ChatItem)-[:ResponseTo]-(i2:ChatItem)
with u1, i1, i2
match (u2)-[:CreateChat]-(i2)
create (u1)-[:InteractsWith]->(u2)
```

- The above scheme will create an undesirable side effect if a user has responded to her own chatItem, because it will create a self-loop between two users. So, after the first two steps we need to eliminate all self-loops involving the edge "InteractsWith".

```
match (u1)-[r:InteractsWith]->(u1) delete r
```

For each of these neighbors, we need to find

- the number of edges it has with the other members on the same list

```
match (u1:User)-[r1:InteractsWith]->(u2:User)
where u1.id <> u2.id
with u1, collect(u2.id) as neighbors, count(distinct(u2)) as neighborAmount
match (u3:User)-[r2:InteractsWith]->(u4:User)
where (u3.id in neighbors)
AND (u4.id in neighbors)
AND (u3.id <> u4.id)
return u3.id, u4.id, count(r2)
```

- If one member has multiple edges with another member we need to count it as 1 because we care only if the edge exists or not.

```
match (u1:User)-[r1:InteractsWith]->(u2:User)
where u1.id <> u2.id
with u1, collect(u2.id) as neighbors, count(distinct(u2)) as neighborAmount
match (u3:User)-[r2:InteractsWith]->(u4:User)
where (u3.id in neighbors)
AND (u4.id in neighbors)
AND (u3.id <> u4.id)
return u3.id, u4.id, count(r2),
case
```

```

when count(r2) > 0 then 1
else 0
end as value

```

- Last step, combine all steps above together.

```

match (u1:User)-[r1:InteractsWith]->(u2:User)
where u1.id <> u2.id
with u1, collect(u2.id) as neighbors, count(distinct(u2)) as neighborAmount
match (u3:User)-[r2:InteractsWith]->(u4:User)
where (u3.id in neighbors)
AND (u4.id in neighbors)
AND (u3.id <> u4.id)
with u1, u3, u4, neighborAmount,
case
when (u3)-->(u4) then 1
else 0
end as value
return u1, sum(value)*1.0/(neighborAmount*(neighborAmount-1)) as coeff
order by coeff desc limit 10

```

#### Most Active Users (based on Cluster Coefficients)

| User ID | Coefficient        |
|---------|--------------------|
| 209     | 0.9523809523809523 |
| 554     | 0.9047619047619048 |
| 1087    | 0.8                |