# 1. Google Play Store apps and reviews

Mobile apps are everywhere. They are easy to create and can be lucrative. Because of these two factors, more and more apps are being developed. In this notebook, we will do a comprehensive analysis of the Android app market by comparing over ten thousand apps in Google Play across different categories. We'll look for insights in the data to devise strategies to drive growth and retention.

Let's take a look at the data, which consists of two files:

- `apps.csv` : contains all the details of the applications on Google Play. There are 13 features that describe a given app.
- `user_reviews.csv` : contains 100 reviews for each app, most helpful first (https://www.androidpolice.com/2019/01/21/google-play-stores-redesigned-ratings-and-reviews-section-lets-you-easily-filter-by-star-rating/). The text in each review has been pre-processed and attributed with three new features: Sentiment (Positive, Negative or Neutral), Sentiment Polarity and Sentiment Subjectivity.

In [2]:

```python
# Read in dataset
import pandas as pd
apps_with_duplicates = pd.read_csv('datasets/apps.csv')

# Drop duplicates from apps_with_duplicates
apps = apps_with_duplicates.drop_duplicates()

# Print the total number of apps
print('Total number of apps in the dataset = ', len(apps))

# Have a look at a random sample of 5 rows
print(apps.sample(5))
```

```
Total number of apps in the dataset =  9659
      Unnamed: 0                                 App           Category  \
2477        3186                    The Emirates App     TRAVEL_AND_LOCAL
2991        3750                         РИА Новости  NEWS_AND_MAGAZINES
371          426                         Web Browser       COMMUNICATION
6560        7618           Adventure Time Game Wizard                GAME
7596        8704  Instant DP Downloader for Instagram              SOCIAL

      Rating  Reviews  Size    Installals  Type  Price Content Rating  \
2477     4.4    22748  55.0  1,000,000+  Free      0       Everyone
2991     4.5    44274   8.0  1,000,000+  Free      0       Everyone
371      4.2    10965   2.3    500,000+  Free      0       Everyone
6560     4.1     4107  40.0     50,000+  Paid  $4.99       Everyone
7596     4.7       38   4.1      5,000+  Free      0       Everyone

                 Genres      Last Updated Current Ver Android Ver
2477     Travel & Local    August 6, 2018       4.7.1  5.0 and up
2991  News & Magazines    August 6, 2018       4.0.6  4.4 and up
371      Communication  February 2, 2015           2  3.0 and up
6560         Adventure     July 16, 2015       1.2.0  4.1 and up
7596            Social    August 4, 2018       1.0.4  4.2 and up
```

In [3]:
```
%%nose

correct_apps_with_duplicates = pd.read_csv('datasets/apps.csv')

def test_pandas_loaded():
    assert ('pd' in globals()), "pandas is not imported and aliased as specifi
ed in the instructions."

def test_apps_with_duplicates_loaded():
#     correct_apps_with_duplicates = pd.read_csv('datasets/apps.csv')
    assert (correct_apps_with_duplicates.equals(apps_with_duplicates)), "The d
ata was not correctly read into apps_with_duplicates."

def test_duplicates_dropped():
#     correct_apps_with_duplicates = pd.read_csv('datasets/apps.csv')
    correct_apps = correct_apps_with_duplicates.drop_duplicates()
    assert (correct_apps.equals(apps)), "The duplicates were not correctly dro
pped from apps_with_duplicates."

def test_total_apps():
    correct_total_apps = len(correct_apps_with_duplicates.drop_duplicates())
    assert (correct_total_apps == len(apps)), "The total number of apps is inc
orrect. It should equal 9659."
```

Out[3]: 4/4 tests passed

## 2. Data cleaning

Data cleaning is one of the most essential subtask any data science project. Although it can be a very tedious process, it's worth should never be undermined.

By looking at a random sample of the dataset rows (from the above task), we observe that some entries in the columns like `Installs` and `Price` have a few special characters ( `+` `,` `$` ) due to the way the numbers have been represented. This prevents the columns from being purely numeric, making it difficult to use them in subsequent future mathematical calculations. Ideally, as their names suggest, we would want these columns to contain only digits from [0-9].

Hence, we now proceed to clean our data. Specifically, the special characters `,` and `+` present in `Installs` column and `$` present in `Price` column need to be removed.

It is also always a good practice to print a summary of your dataframe after completing data cleaning. We will use the `info()` method to acheive this.

In [4]:

```
# List of characters to remove
chars_to_remove = ["+", ",", "$"]
# List of column names to clean
cols_to_clean = ['Installs', 'Price']

# Loop for each column in cols_to_clean
for col in cols_to_clean:
    # Loop for each char in chars_to_remove
    for char in chars_to_remove:
        # Replace the character with an empty string
        apps[col] = apps[col].apply(lambda x: x.replace(char, ''))

# Print a summary of the apps dataframe
print(apps.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 9659 entries, 0 to 9658
Data columns (total 14 columns):
Unnamed: 0        9659 non-null int64
App               9659 non-null object
Category          9659 non-null object
Rating            8196 non-null float64
Reviews           9659 non-null int64
Size              8432 non-null float64
Installs          9659 non-null object
Type              9659 non-null object
Price             9659 non-null object
Content Rating    9659 non-null object
Genres            9659 non-null object
Last Updated      9659 non-null object
Current Ver       9651 non-null object
Android Ver       9657 non-null object
dtypes: float64(2), int64(2), object(10)
memory usage: 1.1+ MB
None
```

In [5]:
```
%%nose
import numpy as np

def test_installs_plus():
    installs = apps['Installs'].values
    plus_removed_correctly = all('+' not in val for val in installs)
    assert plus_removed_correctly, \
    'Some of the "+" characters still remain in the Installs column.'

def test_installs_comma():
    installs = apps['Installs'].values
    comma_removed_correctly = all(',' not in val for val in installs)
    assert comma_removed_correctly, \
    'Some of the "," characters still remain in the Installs column.'

def test_price_dollar():
    prices = apps['Price'].values
    dollar_removed_correctly = all('$' not in val for val in prices)
    assert dollar_removed_correctly, \
    'Some of the "$" characters still remain in the Price column.'
```

Out[5]: 3/3 tests passed

## 3. Correcting data types

From the previous task we noticed that `Installs` and `Price` were categorized as `object` data type (and not `int` or `float` ) as we would like. This is because these two columns originally had mixed input types: digits and special characters. To know more about Pandas data types, read [this (https://datacarpentry.org/python-ecology-lesson/04-data-types-and-format/)](https://datacarpentry.org/python-ecology-lesson/04-data-types-and-format/).

The four features that we will be working with most frequently henceforth are `Installs` , `Size` , `Rating` and `Price` . While `Size` and `Rating` are both `float` (i.e. purely numerical data types), we still need to work on `Installs` and `Price` to make them numeric.

In [6]:
```python
import numpy as np

# Convert Installs to float data type
apps['Installs'] = apps['Installs'].astype('float')

# Convert Price to float data type
apps['Price'] = apps['Price'].astype('float')

# Checking dtypes of the apps dataframe
print(apps.dtypes)
```

```
Unnamed: 0          int64
App                object
Category           object
Rating            float64
Reviews             int64
Size              float64
Installs          float64
Type               object
Price             float64
Content Rating     object
Genres             object
Last Updated       object
Current Ver        object
Android Ver        object
dtype: object
```

In [7]:
```python
%%nose
import numpy as np

def test_installs_numeric():
    assert isinstance(apps['Installs'][0], np.float64), \
    'The Installs column is not of numeric data type (float).'

def test_price_numeric():
    assert isinstance(apps['Price'][0], np.float64), \
    'The Price column is not of numeric data type (float).'
```

Out[7]:
```
2/2 tests passed
```

## 4. Exploring app categories

With more than 1 billion active users in 190 countries around the world, Google Play continues to be an important distribution platform to build a global audience. For businesses to get their apps in front of users, it's important to make them more quickly and easily discoverable on Google Play. To improve the overall search experience, Google has introduced the concept of grouping apps into categories.

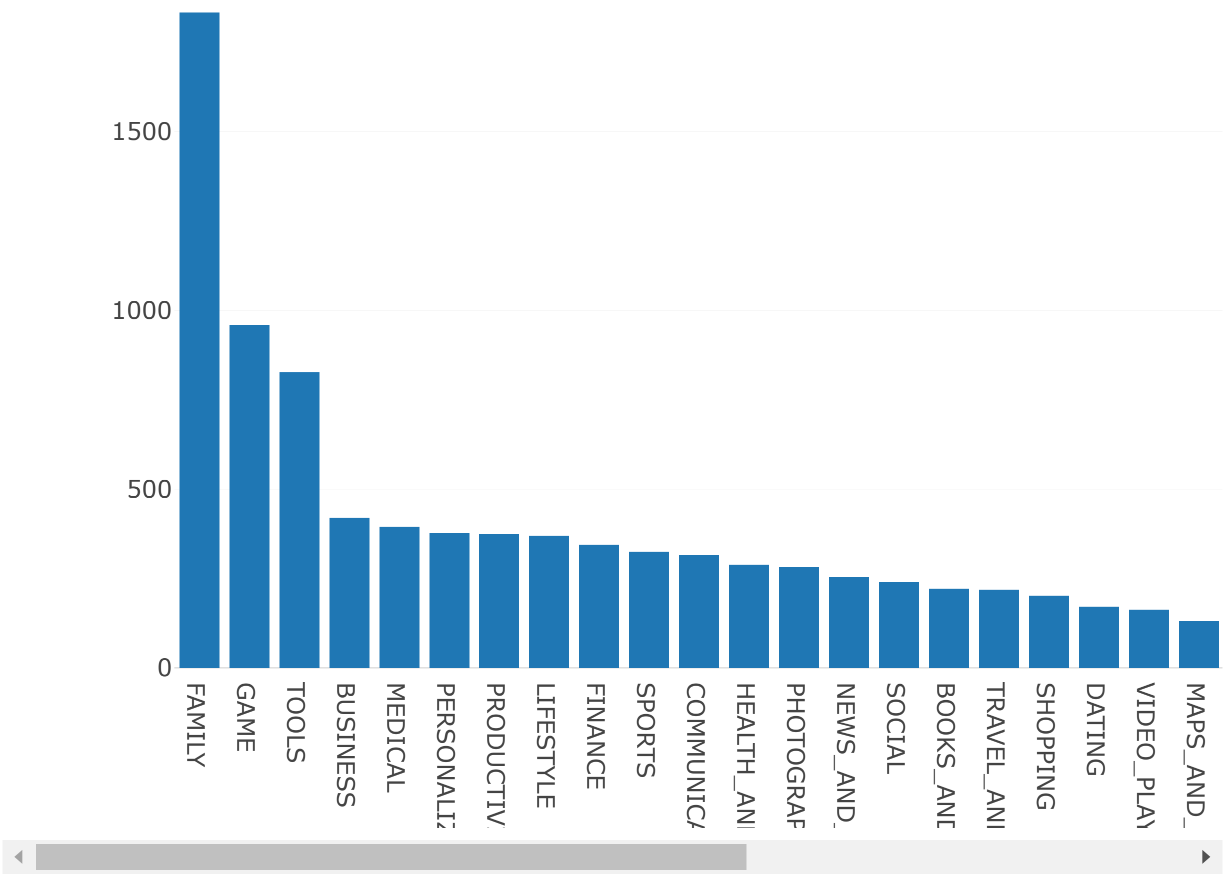
This brings us to the following questions:

- Which category has the highest share of (active) apps in the market?
- Is any specific category dominating the market?
- Which categories have the fewest number of apps?

We will see that there are `33` unique app categories present in our dataset. *Family* and *Game* apps have the highest market prevalence. Interestingly, *Tools*, *Business* and *Medical* apps are also at the top.

In [8]:
```python
import plotly
plotly.offline.init_notebook_mode(connected=True)
import plotly.graph_objs as go

# Print the total number of unique categories
num_categories = len(apps['Category'].unique())
print('Number of categories = ', num_categories)

# Count the number of apps in each 'Category'.
num_apps_in_category = apps['Category'].value_counts()

# Sort num_apps_in_category in descending order based on the count of apps in
 each category
sorted_num_apps_in_category = num_apps_in_category.sort_values(ascending=False
)

data = [go.Bar(
        x = num_apps_in_category.index, # index = category name
        y = num_apps_in_category.values, # value = count
)]

plotly.offline.iplot(data)
```

Number of categories =  33




In [9]:

```
%%nose

def test_num_categories():
    assert num_categories == 33, "The number of app categories is incorrect. I
t should equal 33."

def test_num_apps_in_category():
    correct_sorted_num_apps_in_category = apps['Category'].value_counts().sort
_values(ascending=False)
    assert (correct_sorted_num_apps_in_category == sorted_num_apps_in_categor
y).all(), "sorted_num_apps_in_category is not what we expected. Please inspect
the hint."
```

Out[9]: 2/2 tests passed

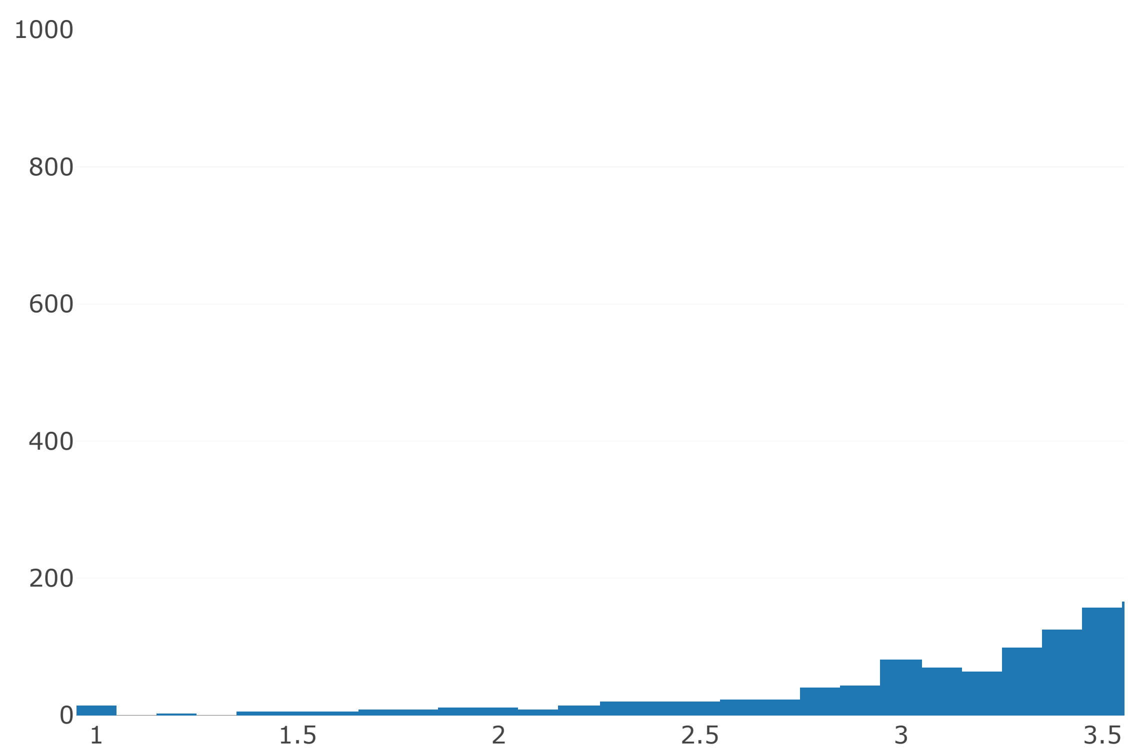
## 5. Distribution of app ratings

After having witnessed the market share for each category of apps, let's see how all these apps perform on an average. App ratings (on a scale of 1 to 5) impact the discoverability, conversion of apps as well as the company's overall brand image. Ratings are a key performance indicator of an app.

From our research, we found that the average volume of ratings across all app categories is `4.17`. The histogram plot is skewed to the left indicating that the majority of the apps are highly rated with only a few exceptions in the low-rated apps.

In [10]:
```
# Average rating of apps
avg_app_rating = np.mean(apps['Rating'])
print('Average app rating = ', avg_app_rating)

# Distribution of apps according to their ratings
data = [go.Histogram(
        x = apps['Rating']
)]

# Vertical dashed line to indicate the average app rating
layout = {'shapes': [{
              'type' :'line',
              'x0': avg_app_rating,
              'y0': 0,
              'x1': avg_app_rating,
              'y1': 1000,
              'line': { 'dash': 'dashdot'}
          }]
          }

plotly.offline.iplot({'data': data, 'layout': layout})
```

```
Average app rating =  4.173243045387994
```



In [11]:

```
%%nose

def test_app_avg_rating():
    assert round(avg_app_rating, 5) == 4.17324, \
    "The average app rating rounded to five digits should be 4.17324."

# def test_x_histogram():
#     correct_x_histogram = apps['Rating']
#     assert correct_x_histogram.all() == data[0]['x'].all(), \
#     'x should equal Rating column'
```

Out[11]: 1/1 tests passed

## 6. Size and price of an app

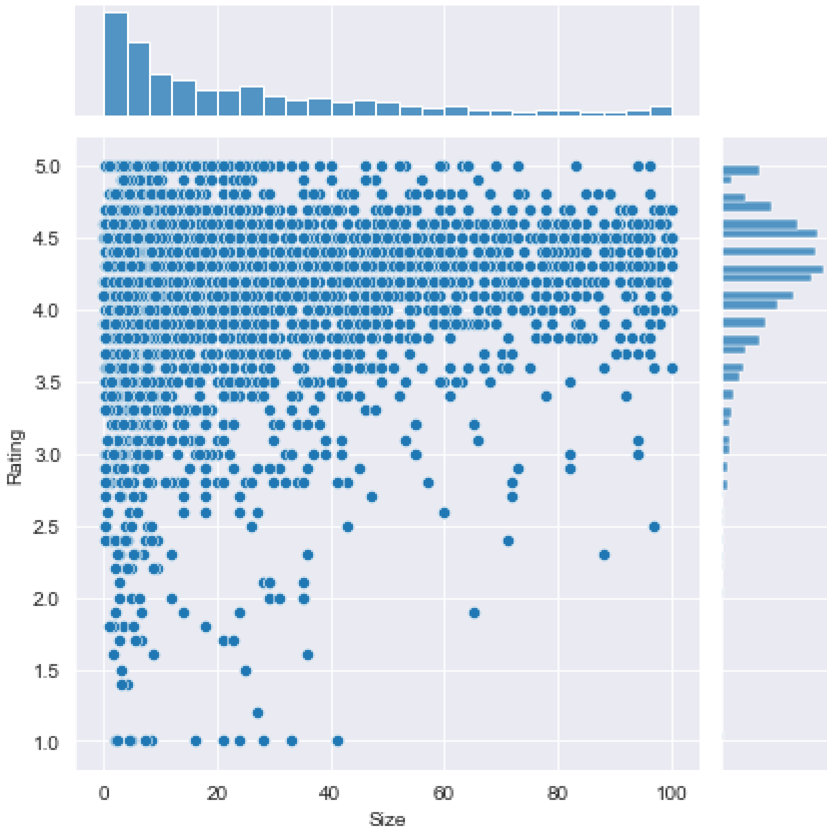
Let's now examine app size and app price. For size, if the mobile app is too large, it may be difficult and/or expensive for users to download. Lengthy download times could turn users off before they even experience your mobile app. Plus, each user's device has a finite amount of disk space. For price, some users expect their apps to be free or inexpensive. These problems compound if the developing world is part of your target market; especially due to internet speeds, earning power and exchange rates.

How can we effectively come up with strategies to size and price our app?

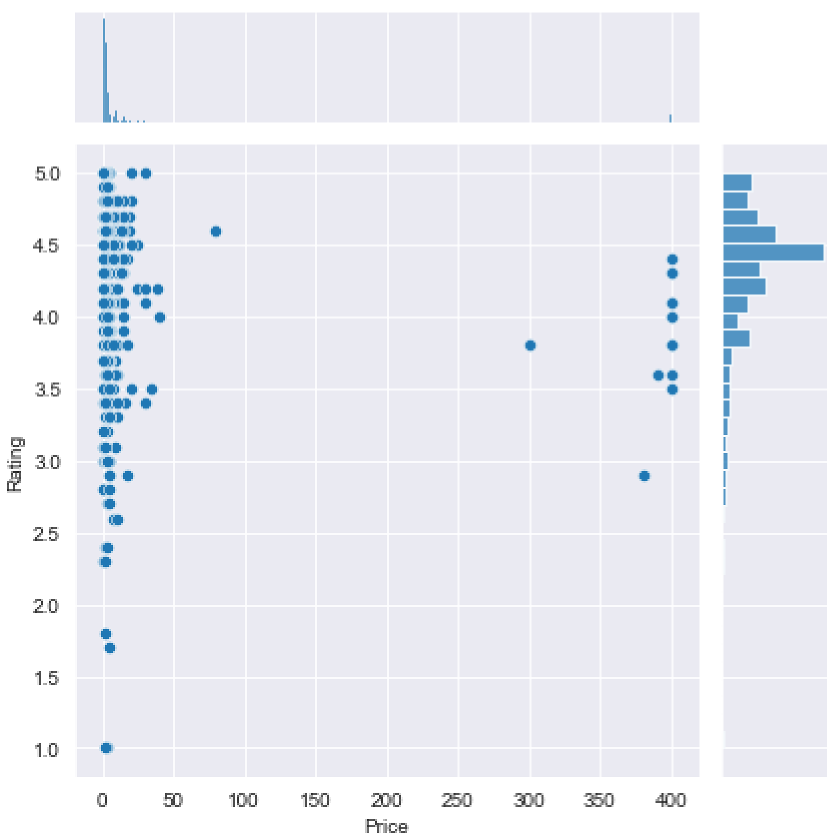
- Does the size of an app affect its rating?
- Do users really care about system-heavy apps or do they prefer light-weighted apps?
- Does the price of an app affect its rating?
- Do users always prefer free apps over paid apps?

We find that the majority of top rated apps (rating over 4) range from 2 MB to 20 MB. We also find that the vast majority of apps price themselves under $10.

In [12]:
```
%matplotlib inline
import seaborn as sns
sns.set_style("darkgrid")
import warnings
warnings.filterwarnings("ignore")

# Select rows where both 'Rating' and 'Size' values are present (ie. the two v
alues are not null)
apps_with_size_and_rating_present = apps[pd.notna(apps['Rating']) & pd.notna(a
pps['Size'])]

# Subset for categories with at least 250 apps
large_categories = apps_with_size_and_rating_present.groupby('Category').filte
r(lambda x: len(x) >= 250)

# Plot size vs. rating
plt1 = sns.jointplot(x = large_categories['Size'], y = large_categories['Ratin
g'])

# Select apps whose 'Type' is 'Paid'
paid_apps = apps_with_size_and_rating_present[apps_with_size_and_rating_presen
t['Type'] == 'Paid']

# Plot price vs. rating
plt2 = sns.jointplot(x = paid_apps['Price'], y = paid_apps['Rating'])
```



5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
Rating
0
20
40
60
80
100
Size

5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
Rating
0
50
100
150
200
250
300
350
400
Price

In [13]:

```
%%nose

correct_apps_with_size_and_rating_present = apps[(~apps['Rating'].isnull()) &
 (~apps['Size'].isnull())]

def test_apps_with_size_and_rating_present():
    global correct_apps_with_size_and_rating_present
    assert correct_apps_with_size_and_rating_present.equals(apps_with_size_and
_rating_present)
    "The correct_apps_with_size_and_rating_present is not what we expected. Pl
ease review the instructions and check the hint if necessary."

def test_large_categories():
    global correct_apps_with_size_and_rating_present
    correct_large_categories = correct_apps_with_size_and_rating_present.group
by(['Category']).filter(lambda x: len(x) >= 250)
    assert correct_large_categories.equals(large_categories), \
    "The large_categories DataFrame is not what we expected. Please review the
instructions and check the hint if necessary."

def test_size_vs_rating():
    global correct_apps_with_size_and_rating_present
    correct_large_categories = correct_apps_with_size_and_rating_present.group
by('Category').filter(lambda x: len(x) >= 250)
#     correct_large_categories = correct_large_categories[correct_large_catego
ries['Size'].notnull()]
#     correct_large_categories = correct_large_categories[correct_large_catego
ries['Rating'].notnull()]
    assert plt1.x.tolist() == large_categories['Size'].values.tolist() and plt
1.y.tolist() == large_categories['Rating'].values.tolist(), \
    "The size vs. rating jointplot is not what we expected. Please review the
 instructions and check the hint if necessary."

def test_paid_apps():
    global correct_apps_with_size_and_rating_present
    correct_paid_apps = correct_apps_with_size_and_rating_present[correct_apps
_with_size_and_rating_present['Type'] == 'Paid']
    assert correct_paid_apps.equals(paid_apps), \
    "The paid_apps DataFrame is not what we expected. Please review the instru
ctions and check the hint if necessary."

def test_price_vs_rating():
    global correct_apps_with_size_and_rating_present
    correct_paid_apps = correct_apps_with_size_and_rating_present[correct_apps
_with_size_and_rating_present['Type'] == 'Paid']
#     correct_paid_apps = correct_paid_apps[correct_paid_apps['Price'].notnull
()]
#     correct_paid_apps = correct_paid_apps[correct_paid_apps['Rating'].notnul
l()]
    assert plt2.x.tolist() == correct_paid_apps['Price'].values.tolist() and p
lt2.y.tolist() == correct_paid_apps['Rating'].values.tolist(), \
    "The price vs. rating jointplot is not what we expected. Please review the
instructions and check the hint if necessary."
```

Out[13]:

```
5/5 tests passed
```

# 7. Relation between app category and app price

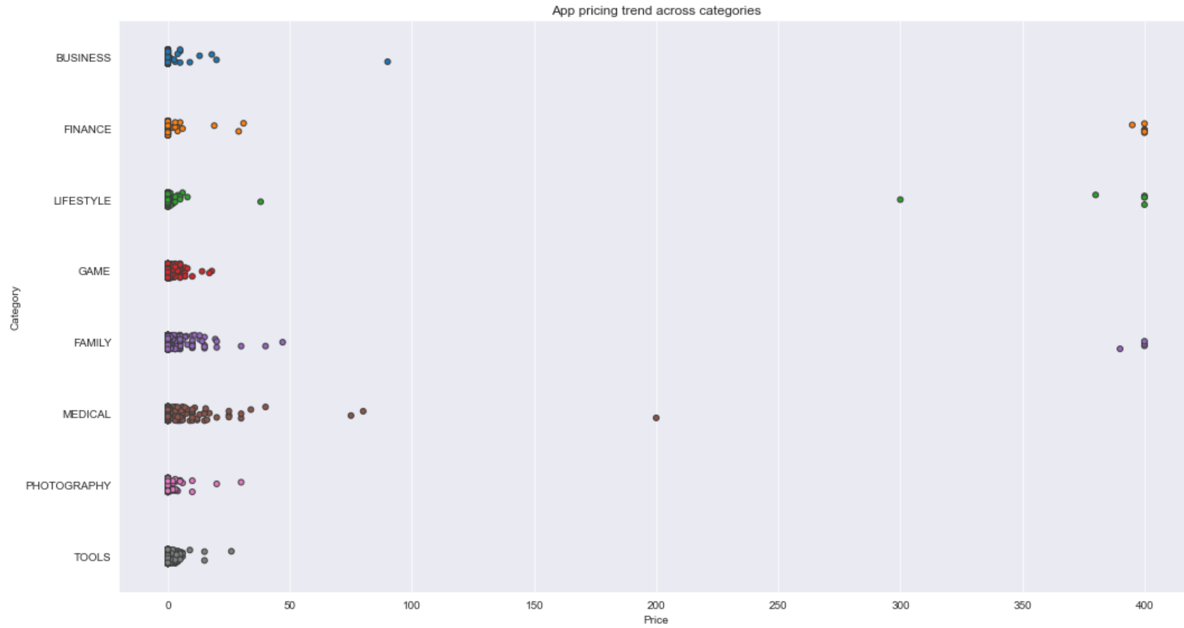
So now comes the hard part. How are companies and developers supposed to make ends meet? What monetization strategies can companies use to maximize profit? The costs of apps are largely based on features, complexity, and platform.

There are many factors to consider when selecting the right pricing strategy for your mobile app. It is important to consider the willingness of your customer to pay for your app. A wrong price could break the deal before the download even happens. Potential customers could be turned off by what they perceive to be a shocking cost, or they might delete an app they've downloaded after receiving too many ads or simply not getting their money's worth.

Different categories demand different price ranges. Some apps that are simple and used daily, like the calculator app, should probably be kept free. However, it would make sense to charge for a highly-specialized medical app that diagnoses diabetic patients. Below, we see that *Medical and Family* apps are the most expensive. Some medical apps extend even up to $80! All game apps are reasonably priced below $20.

In [14]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
fig.set_size_inches(15, 8)

# Select a few popular app categories
popular_app_cats = apps[apps.Category.isin(['GAME', 'FAMILY', 'PHOTOGRAPHY',
                                            'MEDICAL', 'TOOLS', 'FINANCE',
                                            'LIFESTYLE','BUSINESS'])]

# Examine the price trend by plotting Price vs Category
ax = sns.stripplot(x = popular_app_cats['Price'], y = popular_app_cats['Catego
ry'], jitter=True, linewidth=1)
ax.set_title('App pricing trend across categories')
plt.show()

# Apps whose Price is greater than 200
apps_above_200 = apps[apps['Price'] > 200]
apps_above_200[['Category', 'App', 'Price']]
```




Out[14]:

| | Category | App | Price |
|---|---|---|---|
| **3327** | FAMILY | most expensive app (H) | 399.99 |
| **3465** | LIFESTYLE | 💎 I'm rich | 399.99 |
| **3469** | LIFESTYLE | I'm Rich - Trump Edition | 400.00 |
| **4396** | LIFESTYLE | I am rich | 399.99 |
| **4398** | FAMILY | I am Rich Plus | 399.99 |
| **4399** | LIFESTYLE | I am rich VIP | 299.99 |
| **4400** | FINANCE | I Am Rich Premium | 399.99 |
| **4401** | LIFESTYLE | I am extremely Rich | 379.99 |
| **4402** | FINANCE | I am Rich! | 399.99 |
| **4403** | FINANCE | I am rich(premium) | 399.99 |
| **4406** | FAMILY | I Am Rich Pro | 399.99 |
| **4408** | FINANCE | I am rich (Most expensive app) | 399.99 |
| **4410** | FAMILY | I Am Rich | 389.99 |
| **4413** | FINANCE | I am Rich | 399.99 |
| **4417** | FINANCE | I AM RICH PRO PLUS | 399.99 |
| **8763** | FINANCE | Eu Sou Rico | 394.99 |
| **8780** | LIFESTYLE | I'm Rich/Eu sou Rico/أنا غني/我很有錢 | 399.99 |

In [15]:
```
%%nose

last_output = _

def test_apps_above_200():
    assert len(apps_above_200) == 17, "There should be 17 apps priced above 20
0 in apps_above_200."
```

Out[15]: 1/1 tests passed

## 8. Filter out "junk" apps

It looks like a bunch of the really expensive apps are "junk" apps. That is, apps that don't really have a purpose. Some app developer may create an app called *I Am Rich Premium* or *most expensive app (H)* just for a joke or to test their app development skills. Some developers even do this with malicious intent and try to make money by hoping people accidentally click purchase on their app in the store.

Let's filter out these junk apps and re-do our visualization.

In [16]:
```
# Select apps priced below $100
apps_under_100 = popular_app_cats[popular_app_cats['Price'] < 100]

fig, ax = plt.subplots()
fig.set_size_inches(15, 8)

# Examine price vs category with the authentic apps (apps_under_100)
ax = sns.stripplot(x = 'Price', y = 'Category', data = apps_under_100, jitter
= True, linewidth = 1)
ax.set_title('App pricing trend across categories after filtering for junk app
s')
plt.show()
```

In [17]:
```
%%nose

def test_apps_under_100():
    correct_apps_under_100 = popular_app_cats[popular_app_cats['Price'] < 100]
    assert correct_apps_under_100.equals(apps_under_100), \
    "The apps_under_100 DataFrame is not what we expected. Please review the i
nstructions and check the hint if necessary."
```

Out[17]:
```
1/1 tests passed
```

## 9. Popularity of paid apps vs free apps

For apps in the Play Store today, there are five types of pricing strategies: free, freemium, paid, paymium, and subscription. Let's focus on free and paid apps only. Some characteristics of free apps are:

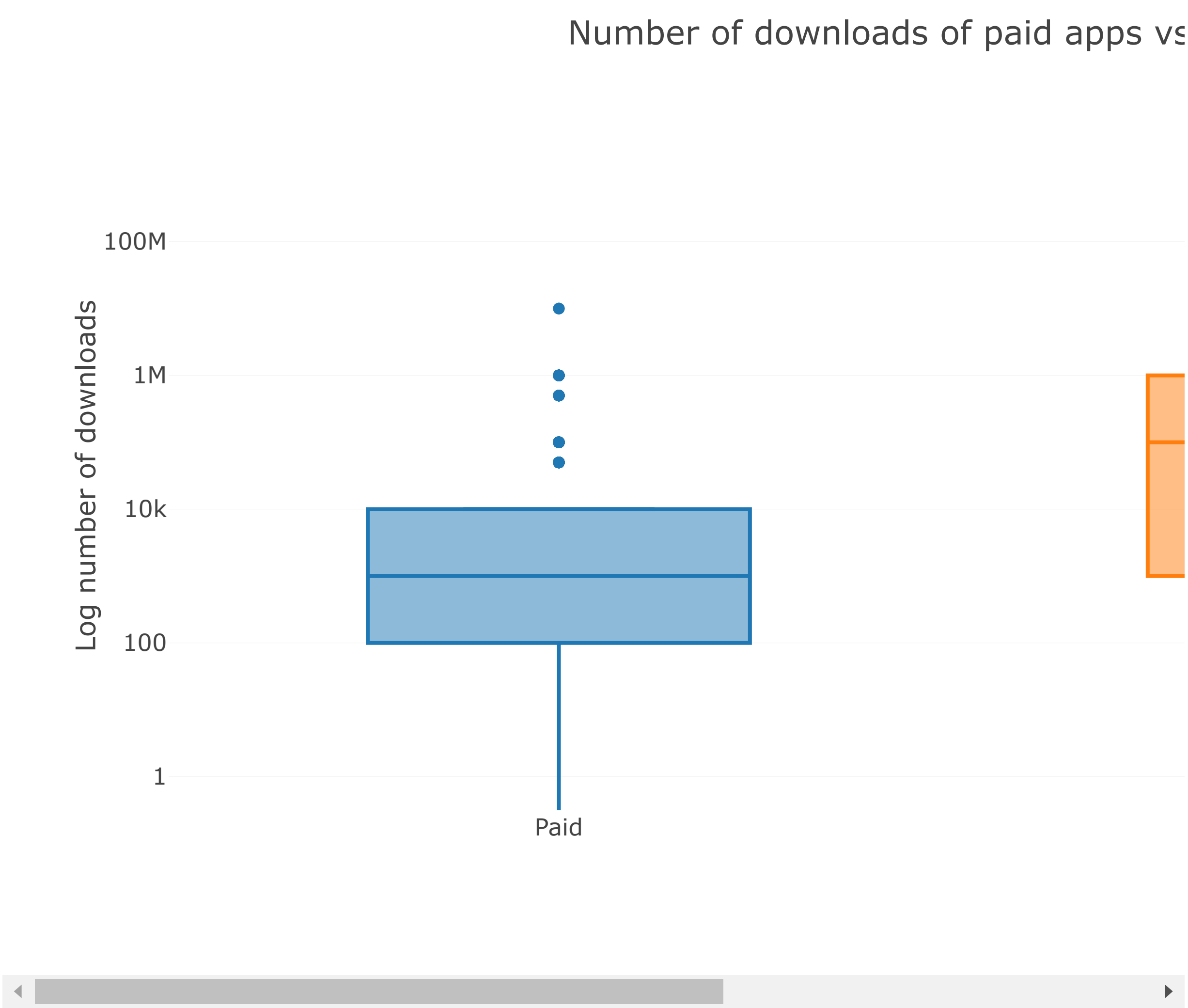
- Free to download.
- Main source of income often comes from advertisements.
- Often created by companies that have other products and the app serves as an extension of those products.
- Can serve as a tool for customer retention, communication, and customer service.

Some characteristics of paid apps are:

- Users are asked to pay once for the app to download and use it.
- The user can't really get a feel for the app before buying it.

Are paid apps installed as much as free apps? It turns out that paid apps have a relatively lower number of installs than free apps, though the difference is not as stark as I would have expected!

In [18]:
```
trace0 = go.Box(
    # Data for paid apps
    y = apps[apps['Type'] == 'Paid']['Installs'],
    name = 'Paid'
)

trace1 = go.Box(
    # Data for free apps
    y = apps[apps['Type'] == 'Free']['Installs'],
    name = 'Free'
)

layout = go.Layout(
    title = "Number of downloads of paid apps vs. free apps",
    yaxis = dict(title = "Log number of downloads",
                type = 'log',
                autorange = True)
)

# Add trace0 and trace1 to a list for plotting
data = [trace0, trace1]
plotly.offline.iplot({'data': data, 'layout': layout})
```



In [19]:
```
%%nose

def test_trace0_y():
    correct_y = apps['Installs'][apps['Type'] == 'Paid']
    assert all(trace0['y'] == correct_y.values), \
    "The y data for trace0 appears incorrect. Please review the instructions a
nd check the hint if necessary."

def test_trace1_y():
    correct_y_1 = apps['Installs'][apps['Type'] == 'Free']
    correct_y_2 = apps['Installs'][apps['Price'] == 0]
    try:
        check_1 = all(trace1['y'] == correct_y_1.values)
    except:
        check_1 = False
    try:
        check_2 = all(trace1['y'] == correct_y_2.values)
    except:
        check_2 = False

    assert check_1 or check_2, \
    "The y data for trace1 appears incorrect. Please review the instructions a
nd check the hint if necessary."
```

Out[19]: 2/2 tests passed

## 10. Sentiment analysis of user reviews

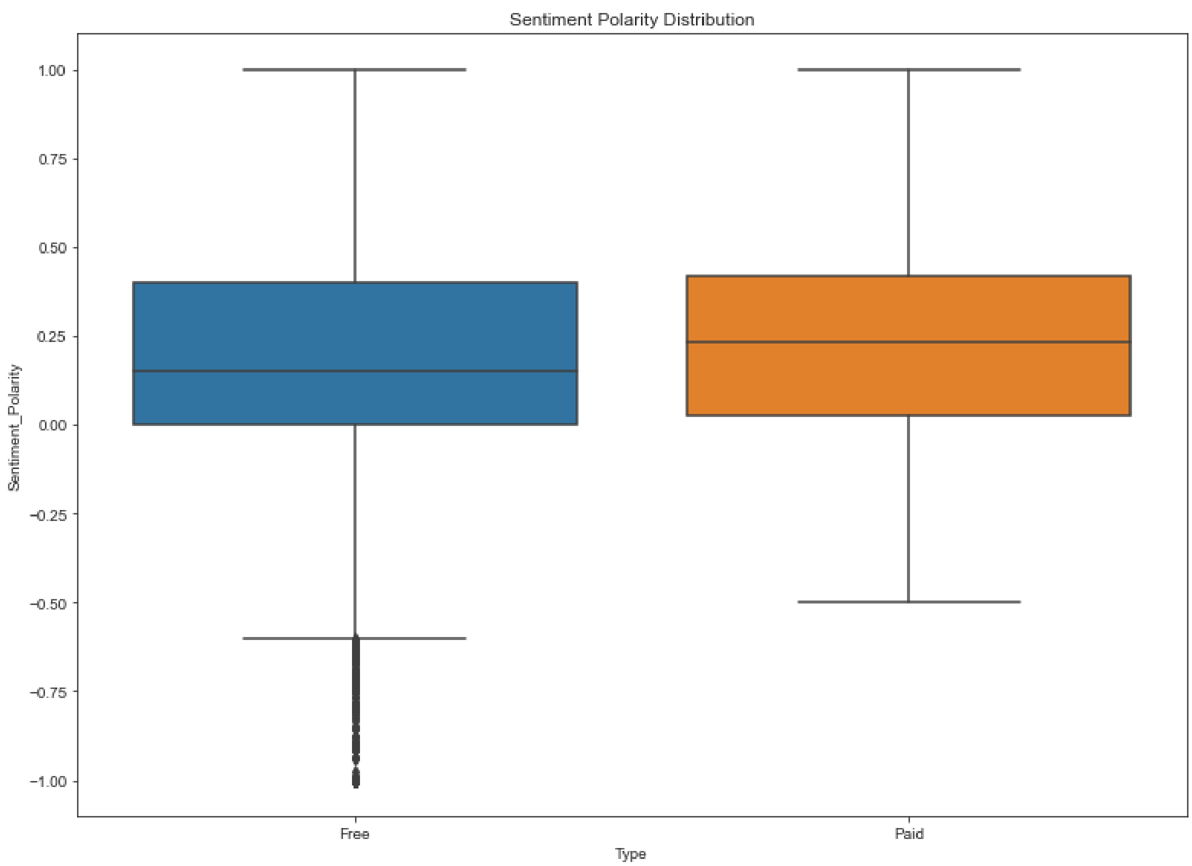
Mining user review data to determine how people feel about your product, brand, or service can be done using a technique called sentiment analysis. User reviews for apps can be analyzed to identify if the mood is positive, negative or neutral about that app. For example, positive words in an app review might include words such as 'amazing', 'friendly', 'good', 'great', and 'love'. Negative words might be words like 'malware', 'hate', 'problem', 'refund', and 'incompetent'.

By plotting sentiment polarity scores of user reviews for paid and free apps, we observe that free apps receive a lot of harsh comments, as indicated by the outliers on the negative y-axis. Reviews for paid apps appear never to be extremely negative. This may indicate something about app quality, i.e., paid apps being of higher quality than free apps on average. The median polarity score for paid apps is a little higher than free apps, thereby syncing with our previous observation.

In this notebook, we analyzed over ten thousand apps from the Google Play Store. We can use our findings to inform our decisions should we ever wish to create an app ourselves.

In [20]:

```
# Load user_reviews.csv
reviews_df = pd.read_csv('datasets/user_reviews.csv')

# Join the two dataframes
merged_df = apps.merge(reviews_df)

# Drop NA values from Sentiment and Review columns
merged_df = merged_df.dropna(subset = ['Sentiment', 'Review'])

sns.set_style('ticks')
fig, ax = plt.subplots()
fig.set_size_inches(11, 8)

# User review sentiment polarity for paid vs. free apps
ax = sns.boxplot(x = 'Type', y = 'Sentiment_Polarity', data = merged_df)
ax.set_title('Sentiment Polarity Distribution')
plt.show()
```



In [21]:

```
%%nose

def test_user_reviews_loaded():
    correct_user_reviews = pd.read_csv('datasets/user_reviews.csv')
    assert (correct_user_reviews.equals(reviews_df)), "The user_reviews.csv file was not correctly loaded. Please review the instructions and inspect the hint if necessary."

def test_user_reviews_merged():
    user_reviews = pd.read_csv('datasets/user_reviews.csv')
    correct_merged = pd.merge(apps, user_reviews, on = "App")
    correct_merged = correct_merged.dropna(subset=['Sentiment', 'Review'])
    assert (correct_merged.equals(merged_df)), "The merging of user_reviews and apps is incorrect. Please review the instructions and inspect the hint if necessary."

def test_project_reset():
    user_reviews = pd.read_csv('datasets/user_reviews.csv')
    assert ('Translated_Reviews' not in user_reviews.columns), "There is an update in the project and some column names have been changed. Please choose the \"Reset Project\" option to fetch the updated copy of the project."
```

Out[21]: 3/3 tests passed