COGS 108 - Final Project

1. Overview

A popular conception is that mainstream, high-budget cultural productions are not as artistically driven as other, more independent projects. However, an equally valid idea is that a larger budget allows for greater effort to be put into a final product. When examined through the more limited scope of films, we decided to examine whether or not a high budget correlates with hightened quality of the movie, as defined by its critical acclaim.

2. Group Members

- Jacques Chazelle: A13059069 (Dataset, Setup)
- Phillip Kim: A13483074 (Data Analysis and Results, Discussion)
- Alex Rickard: A13543393 (Overview, Background and Prior Work, Data Cleaning)
- Elias Rose: A14587288 (Analysis Writeup, Ethics and Privacy, Conclusion)

3. Research Question

Does a Movie’s Budget Correlate with Critical Acclaim?

4. Background and Prior Work

This topic is of interest to our group due to the fact that we all like movies. We were interested in seeing how the perceived quality of a movie compared to its status as a high-production and high-visibility affair.

Most articles and sources that discuss this topic mostly address how a critical score is connected to its box office performance. Rather than looking at how the quality of a film affects its finances, we wanted to look at the question from the other direction: can you buy artistic quality? The numbers show that the highest budget films have wildly varying critical scores, but we believe that this will not hold true when looking at a wider data set. A case study of movie critics notes the fact that movie critics are the arbiters of a movie's quality, which allows us to use an accumulation of their ratings as an accurate benchmark of artistic value.

In one article with a similar study, researchers attempted to predict if a movie would be a flop or a success. The goal of the study was to use machine learning techniques to predict the outcome. They focused on many different variables in this study including cast list, budget, runtime, directors, etc. On a scale that measured success, their machine learning technique predicted a rating of 6 out of a scale of 10 on average. Some films that should have been predicted with a rating higher than 6 would sometimes receive a lower rating. This skewed some of their results for predicting the success of a movie. The article wanted to see what affects movie success with similar goals to our study

References:

1. https://www.the-numbers.com/movie/budgets (https://www.the-numbers.com/movie/budgets)
2. https://www.jstor.org/stable/3569753?seq=1#metadata_info_tab_contents (https://www.jstor.org/stable/3569753?seq=1#metadata_info_tab_contents)
3. http://cs229.stanford.edu/proj2013/cocuzzowu-hitorflop.pdf (http://cs229.stanford.edu/proj2013/cocuzzowu-hitorflop.pdf)

5. Hypothesis

We hypothesize that a movie's critical performance will have diminishing returns when connected to the movie's budget. After a certain point, throwing more money at a movie will not make it "better". We will be comparing a movie's budget with accumulated critic ratings as well as the movies' performance at award shows.

6. Dataset

Our base dataset is sourced from here (https://data.world/popculture/imdb-5000-movie-dataset (also hosted on many other sites). Credits to Chuan Sun (@sundeepblue on Github) who scraped tons of metadata using a combination of The Numbers (www.the-numbers.com), IMDb (imdb.com), and a Python library called "scrapy".

Notes:

- The file is renamed to `movies_5000.csv`
- The dataset contains an incorrect entry for "Star Wars: Episode VII - The Force Awakens". Its IMDb URL was manually changed from https://www.imdb.com/title/tt5289954/ (https://www.imdb.com/title/tt5289954/) to https://www.imdb.com/title/tt2488496/ (https://www.imdb.com/title/tt2488496/) in `movies_5000.csv`
- A lot of data entries are parsed weirdly, lack context, or appear plain incorrect. The intent of the dataset is to use the IMDb urls to scrape the URLs directly.

7. Setup

7.1 Imports used for data aggregation and collection

In [111]:
```
# used for file
import os

# used for saving progress in case of interruption
import pickle

# used for setting scrape rate
import time

# used for distribution checks
from collections import Counter

# used for similarity checks when matching movie titles
! pip install python-levenshtein --user
import Levenshtein

# check for nan
import math

# used for more data types
import numpy as np

# used for dataframes
import pandas as pd

# used for HTTP requests
import requests

# used for Rotten Tomatoes scraping
! pip install rotten_tomatoes_client --user
import rotten_tomatoes_client

# used for generic scraping
from bs4 import BeautifulSoup
```

```
Requirement already satisfied: python-levenshtein in c:\users\test\appdata\roami
ng\python\python36\site-packages (0.12.0)
Requirement already satisfied: setuptools in c:\program files (x86)\microsoft vi
sual studio\shared\anaconda3_64\lib\site-packages (from python-levenshtein)
(39.1.0)
Requirement already satisfied: rotten_tomatoes_client in c:\users\test\appdata\r
oaming\python\python36\site-packages (0.0.3)
Requirement already satisfied: requests in c:\program files (x86)\microsoft visu
al studio\shared\anaconda3_64\lib\site-packages (from rotten_tomatoes_client)
(2.18.4)
Requirement already satisfied: enum34 in c:\users\test\appdata\roaming\python\py
thon36\site-packages (from rotten_tomatoes_client) (1.1.6)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in c:\program files (x86)\m
icrosoft visual studio\shared\anaconda3_64\lib\site-packages (from requests->rot
ten_tomatoes_client) (3.0.4)
Requirement already satisfied: idna<2.7,>=2.5 in c:\program files (x86)\microsof
t visual studio\shared\anaconda3_64\lib\site-packages (from requests->rotten_tom
atoes_client) (2.6)
Requirement already satisfied: urllib3<1.23,>=1.21.1 in c:\program files (x86)\m
icrosoft visual studio\shared\anaconda3_64\lib\site-packages (from requests->rot
ten_tomatoes_client) (1.22)
Requirement already satisfied: certifi>=2017.4.17 in c:\program files (x86)\micr
osoft visual studio\shared\anaconda3_64\lib\site-packages (from requests->rotten
_tomatoes_client) (2018.4.16)
```

7.2 Constants used for file names, columns, and scraping settings

In [112]:
```
# the original movie dataset (with the one entry change)
MOVIES_READ_FILE = 'movies_5000.csv'

# file we write/update movie data to
MOVIES_WRITE_FILE = 'movies_data.csv'

# files to store errors from scraping each site (for debugging/cleaning purposes)
IMDB_ERRORS_FILE = 'imdb_errors.csv'
RT_ERRORS_FILE = 'rt_errors.csv'
AWARDS_ERRORS_FILE = 'awards_errors.csv'

# set minimum delay to not overwhelm site when scraping
SCRAPE_DELAY_S = 1

# scraping modes:
# - "return": returns the data if it exists; skips scraping
# - "resume": resumes scraping progress (starts from beginning if not started)
# - "restart": restart scraping from beginning and overwrite data
SCRAPE_MODES = {'return', 'resume', 'restart'}

# column order in final dataset
ORDERED_COLUMNS = [
    'title', 'year', 'movie_id', 'imdb_url', 'budget_currency', 'budget_amount',
    'metacritic', 'rotten_tomatoes', 'award_nominations', 'award_wins'
]
```

7.3 Helper functions

7.3.1 `scrape()`: generic scrape function that handles scraped data, erroroneous data, and exceptions

```
In [113]: def scrape(df_movies, site_name, scrape_call, save_file, error_file, mode='resume
          '):
              wrapped_scrape_call = wrap_scrape_call(scrape_call)

              site_name_snake_case = site_name.lower().replace(' ', '_')

              start_index = 0
              if mode == 'resume' and os.path.exists('%s.p' % site_name_snake_case):
                  start_index = load_progress(site_name_snake_case)[1]

              # scrape call
              (end_index, error_indices) = wrapped_scrape_call(df_movies, site_name, start_i
          ndex)

              save_progress((site_name_snake_case, max(end_index, 0)))

              df_errors = df_movies.loc[error_indices, :]
              if error_file and os.path.exists(error_file) and mode == 'resume':
                  df_errors = pd.read_csv(error_file, dtype={'movie_id': object}, index_co
          l=0)\
                      .append(df_errors, sort=False)
              else:
                  df_errors['movie_index'] = df_errors.index.values

              df_errors.to_csv(error_file)
              df_movies.to_csv(save_file)

              # write scraping work done so far
              return True if end_index == -1 else False
```

7.3.2 `wrap_scrape_call()`: decorator function that prints scraping progress such that - = 1%, * = 10%, and stores bad entries and progress in case of Exception.

e.g. [---------*-- indicates 12% scraping completion

```
In [114]: def wrap_scrape_call(scrape_call):
              def wrapper(df_movies, scrape_site, start_index):

                  def get_checkpoints():
                      checkpoint_stages = []
                      num_partitions = 100
                      for index in range(1, num_partitions):
                          if index % 10 == 0:
                              checkpoint_stages.append((
                                  index, int(len(df_movies) * index / num_partitions), '*'
                              ))
                          else:
                              checkpoint_stages.append((
                                  index, int(len(df_movies) * index / num_partitions), '-'
                              ))

                      return sorted(checkpoint_stages, reverse=True)

                  checkpoint_stages = get_checkpoints()
                  error_indices = []

                  print('\n%s Scraping Progress starting at index %s:\n* = 10%% - = 1%% ['
                        % (scrape_site, start_index), end='')
                  elapsed_time = time.perf_counter()
                  for index in range(start_index):
                      while checkpoint_stages and index >= checkpoint_stages[-1][1]:
                          print(checkpoint_stages.pop()[2], end='')
                  for index in range(start_index, len(df_movies)):
                      while checkpoint_stages and index >= checkpoint_stages[-1][1]:
                          print(checkpoint_stages.pop()[2], end='')

                      try:
                          start_time = time.perf_counter()
                          scrape_call(df_movies, index, error_indices)
                          elapsed_time = time.perf_counter() - start_time
                          time.sleep(max(0.0, SCRAPE_DELAY_S - elapsed_time))

                      except requests.exceptions.RequestException as e:
                          print('%s]\n' % ('/' * (len(checkpoint_stages) + 1)))
                          print('ERROR at index %s: %s' % (index, e))
                          print('Time Elapsed: %s s' % round(time.perf_counter() - elapsed_t
          ime))

                          return index, error_indices

                  print('*]\n')
                  print('Time Elapsed: %s s' % round(time.perf_counter() - elapsed_time))

                  return -1, error_indices

              return wrapper
```

7.3.3 `save_progress(), load_progress()`: pair of progress saving/loading functions used in case of Exception midway through scraping

```
In [115]: def save_progress(site_index_tuple):
              pickle.dump(site_index_tuple, open('%s.p' % site_index_tuple[0], 'wb'))


          def load_progress(site_name):
              return pickle.load(open('%s.p' % site_name, 'rb'))
```

7.4 Code Overview

The code to build the data file is structured as follows:

1. Read the movies file `movies_5000.csv` and save the dataframe to `movies_data.csv`.
2. Scrape IMDb for updated movie and Metacritic info.
3. IMDb scraping returned quite a few errors. Rescrape to find what stands out these movies and drop as necessary. *Spoilers: these are actually TV show entries.*
4. Scrape IMDb award URL for award counts.
5. Scrape Rotten Tomatoes using an external library.
6. Reorganize the final movie data file.

If at any point there is an error when scraping, progress issaved and the program halts, skipping subsequent steps.

```
In [116]: # builds dataset using 5000 movies list; default is to NOT overwrite old data
          def build_movie_file():
              # build dataset
              read_movies(mode='return')

              if not add_imdb_info(mode='return'):
                  return

              if not add_imdb_errors_info(mode='return'):
                  return

              if not add_award_info(mode='return'):
                  return

              if not add_rt_info(mode='return'):
                  return

              if not organize_file(mode='return'):
                  return
```

7.5 Reading the Movie File

Below are the relevant functions for reading the original movie file. The columns and rows are cleaned up for the sake of later use. The movie_id column is added as a unique identifier for each movie.

7.5.1 `read_movies()`: function called by `build_movie_file()` that handles organizing the read data.

```
In [117]: def read_movies(mode='return'):

              # assert scrape mode
              if mode not in SCRAPE_MODES:
                  raise ValueError('mode=%s must be from %s' % (mode, SCRAPE_MODES))

              # do not read raw data if file exists
              if mode == 'return' and os.path.exists(MOVIES_WRITE_FILE):
                  print('\nFound movies_data.csv: skip reading movies_5000.csv')
                  return pd.read_csv(MOVIES_WRITE_FILE, dtype={'movie_id': object}, index_co
          l=0)

              # return fresh file
              df_movies = pd.read_csv(MOVIES_READ_FILE)
              assert len(df_movies) == 5043 and len(list(df_movies)) == 28

              # keep [11, 17, 22, 23]: movie_title, movie_imdb_link, budget, title_year
              indices_to_drop = [
                  *range(0, 11), *range(12, 17), *range(18, 22), *range(24, len(list(df_movi
          es)))
              ]
              df_movies.drop(df_movies.columns[indices_to_drop], axis=1, inplace=True)
              assert list(df_movies) == ['movie_title', 'movie_imdb_link', 'budget', 'title_
          year']

              # rename columns
              df_movies.rename({
                  'movie_title': 'title',
                  'movie_imdb_link': 'imdb_url',
                  'budget': 'budget_amount',
                  'title_year': 'year'
              }, axis=1, inplace=True)
              assert list(df_movies) == ['title', 'imdb_url', 'budget_amount', 'year']

              # drop duplicate rows
              df_movies.drop_duplicates(subset=['imdb_url'], inplace=True)
              df_movies.reset_index(drop=True, inplace=True)
              assert len(df_movies) == 4919

              # add unique imdb movie id
              add_movie_id(df_movies)

              df_movies.to_csv(MOVIES_WRITE_FILE)
```

7.5.2 `add_movie_id()`: adds *movie_id* column and ensures each row's entry is unique.

```
In [118]: def add_movie_id(df_movies):

              # extract movie ID from imdb link
              df_movies['movie_id'] = [
                  extract_movie_id(imdb_url) for imdb_url in df_movies['imdb_url']
              ]

              # remove useless trailing url text
              df_movies['imdb_url'] = [
                  df_movies.at[i, 'imdb_url']
                  [:df_movies.at[i, 'imdb_url'].find(df_movies.at[i, 'movie_id']) +
                      len(df_movies.at[i, 'movie_id']) + 1]
                  for i in range(len(df_movies))
              ]

              # ensure each movie has a proper movie ID
              movie_id_distribution = Counter(df_movies['movie_id'])
              assert len(movie_id_distribution) == len(df_movies)
              assert list(df_movies) == ['title', 'imdb_url', 'budget_amount', 'year', 'movi
          e_id']
```

7.5.3 `extract_movie_id()`: extracts movie_id from the URL that contains it.

```
In [119]: def extract_movie_id(imdb_url):
              title_url = '/title/tt'
              start_index = imdb_url.find(title_url) + len(title_url)
              end_index = imdb_url.find('/', start_index)

              return imdb_url[start_index:end_index]
```

7.6 Adding IMDb Info

Each movie is scraped using BeautifulSoup and matched using the unique IMDb URL.

7.6.1 `add_imdb_info()`: function called by `build_movie_file()` that handles how to scrape the site and parse the HTML. Adds up to date imdb info: title, year, budget (currency and amount), and Metacritic rating.

```
In [120]: def add_imdb_info(mode='return'):

              # assert scrape mode
              if mode not in SCRAPE_MODES:
                  raise ValueError('mode=%s must be from %s' % (mode, SCRAPE_MODES))

              df_movies = pd.read_csv(MOVIES_WRITE_FILE, dtype={'movie_id': object}, index_c
          ol=0)

              # do not re-extract imdb info if exists
              if mode == 'return':
                  mode = 'resume'
                  if 'metacritic' in list(df_movies):
                      print('IMDb info already added: skipping scraping')
                      return True

              # set distinct default values for columns
              if mode == 'restart' or 'metacritic' not in list(df_movies):
                  df_movies['budget_currency'] = ''
                  df_movies['metacritic'] = -1

              # get info from imdb
              def scrape_imdb(df_movies, index, error_indices):

                  page = requests.get(df_movies.at[index, 'imdb_url'])
                  soup = BeautifulSoup(page.content, 'html.parser')

                  try:
                      (
                          df_movies.at[index, 'title'],
                          df_movies.at[index, 'year']
                      ) = separate_title_year(soup.find('h1', class_='').text)
                  except AttributeError:
                      pass
                  except ValueError:
                      error_indices.append(index)

                  try:
                      (
                          df_movies.at[index, 'budget_currency'],
                          df_movies.at[index, 'budget_amount']
                      ) = separate_currency_amount(
                          soup.find('h4', class_='inline', text='Budget:').next_sibling.stri
          p()
                      )
                  except AttributeError:
                      pass
                  except ValueError:
                      error_indices.append(index)

                  try:
                      df_movies.at[index, 'metacritic'] = \
                          int(soup.find('div', class_='metacriticScore').text.strip())
                  except AttributeError:
                      pass

              return True if scrape(
                  df_movies, 'IMDb', scrape_imdb, MOVIES_WRITE_FILE, IMDB_ERRORS_FILE, mode=
          mode
              ) else False
```

7.6.2 `separate_title_year(), separate_currency_amount()`: helper functions to separate the sought out elements from HTML strings.

```
In [121]: def separate_title_year(in_str):
              year_start_index = in_str.rfind('(') + 1
              year_end_index = in_str.rfind(')')

              return (
                  in_str[:year_start_index - 1].strip(),
                  int(in_str[year_start_index:year_end_index])
              )


          def separate_currency_amount(in_str):
              amount_start_index = [c.isdigit() for c in in_str].index(True)

              return (
                  in_str[:amount_start_index].strip(),
                  int(in_str[amount_start_index:].replace(',', ''))
              )
```

7.7 Checking IMDb Errors

Scraping IMDb returned several error entries. Rescrape these movies and gather more data about them.

7.7.1 `add_imdb_errors_info()`: function called by `build_movie_file()` that handles how to scrape the site and parse the HTML. In particular, the media type is sought.

In [122]:
```
def add_imdb_errors_info(mode='return'):

    # assert scrape mode
    if mode not in SCRAPE_MODES:
        raise ValueError('mode=%s must be from %s' % (mode, SCRAPE_MODES))

    # read imdb errors file
    df_errors = pd.read_csv(IMDB_ERRORS_FILE, dtype={'movie_id': object}, index_co
l=0)

    # do not re-extract imdb info if exists
    if mode == 'return':
        mode = 'resume'
        if 'type' in list(df_errors):
            print('IMDb Errors info already added: skipping scraping')
            return True

    # set distinct default values for columns
    if mode == 'restart' or 'type' not in df_errors:
        df_errors['type'] = ''

    if 'movie_index' not in df_errors:
        df_errors['movie_index'] = df_errors.index.values
        df_errors.reset_index(drop=True, inplace=True)

    # get info from imdb for errors
    def scrape_imdb_errors(df_movies, index, error_indices):
        page = requests.get(df_movies.at[index, 'imdb_url'])
        soup = BeautifulSoup(page.content, 'html.parser')

        try:
            df_movies.at[index, 'title'] = soup.find('h1', class_='').text.strip()
        except AttributeError:
            pass
        except ValueError:
            error_indices.append(index)

        try:
            df_movies.at[index, 'type'] = fix_imdb_error_type(soup.find(
                'a',
                title='See more release dates'
            ).text.strip())
        except AttributeError:
            pass
        except ValueError:
            error_indices.append(index)

    if not scrape(
            df_errors, 'IMDb errors', scrape_imdb_errors, IMDB_ERRORS_FILE, None,
mode=mode
    ):
        return False

    print('Errors found for types: %s\n' % df_errors['type'].unique())

    df_movies = pd.read_csv(MOVIES_WRITE_FILE, dtype={'movie_id': object}, index_c
ol=0)
    df_movies = df_movies.drop(list(df_errors['movie_index'].values))
    df_movies.reset_index(drop=True, inplace=True)
    df_movies.to_csv(MOVIES_WRITE_FILE)
    print('Removed the above types from movies_data.csv\n')
```

7.7.2 `fix_imdb_error_type()`: extracts the media type from the HTML string.

In [123]:
```
def fix_imdb_error_type(in_str):
    end_index = in_str.rfind(' (')

    if end_index != -1:
        return in_str[:end_index]

    end_index = in_str.find(' aired')
    if end_index != -1:
        return in_str[:end_index]
```

7.8 Adding IMDb Award Info

IMDb has information about movies' awards that can be accessed by simply appending "awards/" to the movie url.

7.8.1 `add_award_info()`: function called by build_movie_file() that handles how to scrape the site and parse the HTML. Adds number of award nominations and wins.

```
In [124]: def add_award_info(mode='return'):

              # assert scrape mode
              if mode not in SCRAPE_MODES:
                  raise ValueError('mode=%s must be from %s' % (mode, SCRAPE_MODES))

              df_movies = pd.read_csv(MOVIES_WRITE_FILE, dtype={'movie_id': object}, index_c
          ol=0)

              # do not re-extract imdb info if exists
              if mode == 'return':
                  mode = 'resume'
                  if 'award_nominations' in list(df_movies):
                      print('Award info already added: skipping scraping')
                      return True

              # set distinct default values
              if mode == 'restart' or 'award_nominations' not in list(df_movies):
                  df_movies['award_nominations'] = 0
                  df_movies['award_wins'] = 0

              # get info from imdb award pages
              def scrape_awards(df_movies, index, error_indices):

                  page = requests.get('%sawards/' % df_movies.at[index, 'imdb_url'])
                  soup = BeautifulSoup(page.content, 'html.parser')

                  try:
                       (
                           df_movies.at[index, 'award_nominations'],
                           df_movies.at[index, 'award_wins']
                       ) = separate_award_nominations_wins(soup.find('div', class_='desc').te
          xt)
                  except AttributeError:
                      pass
                  except ValueError:
                      error_indices.append(index)

              return True if scrape(
                  df_movies, 'IMDb Awards', scrape_awards, MOVIES_WRITE_FILE, AWARDS_ERRORS_
          FILE,
                  mode=mode
              ) else False
```

7.8.2 `separate_award_nominations_wins()`: separates the award nominations and wins within the HTML string.

```
In [125]: def separate_award_nominations_wins(in_str):
              pre_win_str = 'all '
              post_win_str = ' win'
              wins_start_index = in_str.find(pre_win_str) + len(pre_win_str)
              wins_end_index = in_str.find(post_win_str, wins_start_index)

              pre_nom_str = 'and '
              post_nom_str = ' nomination'
              nom_start_index = in_str.find(pre_nom_str, wins_end_index + len(post_win_st
          r))\
                                 + len(pre_nom_str)
              nom_end_index = in_str.find(post_nom_str, nom_start_index)

              return (
                  int(in_str[nom_start_index:nom_end_index]),
                  int(in_str[wins_start_index:wins_end_index])
              )
```

7.9 Adding Rotten Tomatoes Rating Info

Scraping Rotten Tomatoes is more difficult than IMDb as unlike for IMDb, the URLs are neither in the original dataset nor easy to consistently guess. Rotten Tomatoes URLs depend of the title and year of the movie, which can vary from IMDb data. Instead of guessing until a page is found, we use a third party library that allows for querying for results, from whcih we can decide the "best" result.

7.9.1 `add_rt_info()`: function called by build_movie_file() that handles how to use rotten_tomatoes_client, pass a query, and choose results.

In [126]:
```
def add_rt_info(mode='return'):

    # assert scrape mode
    if mode not in SCRAPE_MODES:
        raise ValueError('mode=%s must be from %s' % (mode, SCRAPE_MODES))

    df_movies = pd.read_csv(MOVIES_WRITE_FILE, dtype={'movie_id': object}, index_c
ol=0)

    # do not re-extract imdb info if exists
    if mode == 'return':
        mode = 'resume'
        if 'rotten_tomatoes' in list(df_movies):
            return True

    # set distinct default values for columns
    if mode == 'restart' or 'rotten_tomatoes' not in list(df_movies):
        df_movies['rotten_tomatoes'] = -1

    # scrape Rotten Tomatoes
    rt_client = rotten_tomatoes_client.RottenTomatoesClient()

    def scrape_rt(df_movies, index, error_indices):

        # since no way to get an exact match, use criteria like year and closest t
itle
        possible_choices = []
        for movie in rt_client.search(df_movies.at[index, 'title'])['movies']:
            # filter movie choice that do not have matching years
            if movie['year'] \
                    and not math.isnan(df_movies.at[index, 'year']) \
                    and abs(movie['year'] - df_movies.at[index, 'year']) > 5:
                continue

            possible_choices.append(movie)

        # no choices found, set distinct value
        if len(possible_choices) == 0:
            df_movies.at[index, 'rotten_tomatoes'] = -2
            error_indices.append(index)
            return

        # multiple choice found, choose closest name
        if len(possible_choices) > 1:
            title_similarities = [
                Levenshtein.ratio(possible_choice['name'], df_movies.at[index, 'ti
tle'])
                for possible_choice in possible_choices
            ]
            possible_choices = [
                possible_choices[title_similarities.index(max(title_similaritie
s))]
            ]

        best_choice = possible_choices[0]
        if 'meterScore' in best_choice:
            df_movies.at[index, 'rotten_tomatoes'] = best_choice['meterScore']

    return True if scrape(
        df_movies, 'RT', scrape_rt, MOVIES_WRITE_FILE, RT_ERRORS_FILE, mode=mode
    ) else False
```

7.10 Organzing the Final Dataset

For the sake of clarity, the final movie file has its columns rearranged, types changed, and NaN values substituted.

7.10.1 `organize_file()`: reorganizes movie file unless already done.

```
In [127]: def organize_file(mode='return'):

              # assert scrape mode
              if mode not in SCRAPE_MODES:
                  raise ValueError('mode=%s must be from %s' % (mode, SCRAPE_MODES))

              df_movies = pd.read_csv(MOVIES_WRITE_FILE, dtype={'movie_id': object}, index_c
          ol=0)

              if mode == 'return' \
                      and list(df_movies.columns) == ORDERED_COLUMNS \
                      and df_movies['year'].dtype == np.int64 \
                      and df_movies['budget_amount'].dtype == np.int64 \
                      and not df_movies.isna().values.any():
                  print('%s already organized' % MOVIES_WRITE_FILE)
                  return

              df_movies = df_movies[ORDERED_COLUMNS]
              df_movies['year'] = df_movies['year'].astype(int)
              df_movies['budget_amount'].fillna(-1, inplace=True)
              df_movies['budget_amount'] = df_movies['budget_amount'].astype(int)
              df_movies['budget_currency'].fillna('UNLISTED', inplace=True)

              df_movies.to_csv(MOVIES_WRITE_FILE)
```

7.11 Running the Code

The movie file is already processed, but let us the run the code again without rescraping (which would take an estimated 8 hours total):

```
In [128]: build_movie_file()
```

```
Found movies_data.csv: skip reading movies_5000.csv
IMDb info already added: skipping scraping
IMDb Errors info already added: skipping scraping
Award info already added: skipping scraping
movies_data.csv already organized
```

Resulting dataset looks like this:

```
In [129]: pd.read_csv(MOVIES_WRITE_FILE, dtype={'movie_id': object}, index_col=0).head()
```

Out[129]:

	title	year	movie_id	imdb_url	...	metacritic	rotten_tomatoes	award_nominations
0	Avatar	2009	0499549	http://www.imdb.com/title/tt0499549/	...	83	82	129
1	Pirates of the Caribbean: At World's End	2007	0449088	http://www.imdb.com/title/tt0449088/	...	50	44	46
2	Spectre	2015	2379713	http://www.imdb.com/title/tt2379713/	...	60	64	34
3	The Dark Knight Rises	2012	1345836	http://www.imdb.com/title/tt1345836/	...	78	87	102
4	Star Wars: Episode VII - The Force Awakens	2015	2488496	http://www.imdb.com/title/tt2488496/	...	81	92	129

5 rows × 10 columns

8. Data Cleaning

```
In [130]: import pandas as pd
          import numpy as np
```

8.1 Cleaning the data

After loading in the data, we remove all rows with null values. This largely focuses on the budget and critical scores columns. We ignore the award-related columns as most movies are not considered for awards and so will not have data in these columns. Following this, we remove rows with null-equivalent values, such as '-1' in the critical rating areas. In addition, we limit the dataset to movies with budgets in American dollars. Though we could research each conversion, the fluctuation of the conversion relationship is too large to get an accurate final value. We then remove any outliers that escaped this by capping the budget amount at the highest American-dollar budget ever. Finally, we remove duplicates from the remaining subset and export the data to a new, cleaned file.

In [131]:
```
df_cleaning = pd.read_csv(MOVIES_WRITE_FILE, dtype={'movie_id': object}, index_co
l=0)
df_cleaning.dropna(subset=['title','metacritic','rotten_tomatoes','budget_amount
'], inplace=True)
df_cleaning = df_cleaning[df_cleaning['metacritic'] > 0]
df_cleaning = df_cleaning[df_cleaning['rotten_tomatoes'] > 0]
df_cleaning = df_cleaning[df_cleaning['budget_amount'] > 0]
df_cleaning = df_cleaning[df_cleaning['year'] > 0]
df_cleaning = df_cleaning[df_cleaning['budget_currency'] == '$']
df_cleaning = df_cleaning[df_cleaning['budget_amount'] < 400000000]
df_cleaning = df_cleaning.drop_duplicates(subset=['movie_id'], keep='first')
df_cleaning.to_csv('movies_data_cleaned.csv')
```

9. Data Analysis & Results

In [132]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [133]:
```
#Configure libraries
sns.set()
sns.set_context('talk')

#Max Rows/Column
pd.options.display.max_rows = 7
pd.options.display.max_columns = 8
```

In [134]:
```
new_df = pd.read_csv('movies_data_cleaned.csv')
new_df.columns
```

Out[134]:
```
Index(['Unnamed: 0', 'title', 'year', 'movie_id', 'imdb_url',
       'budget_currency', 'budget_amount', 'metacritic', 'rotten_tomatoes',
       'award_nominations', 'award_wins'],
      dtype='object')
```

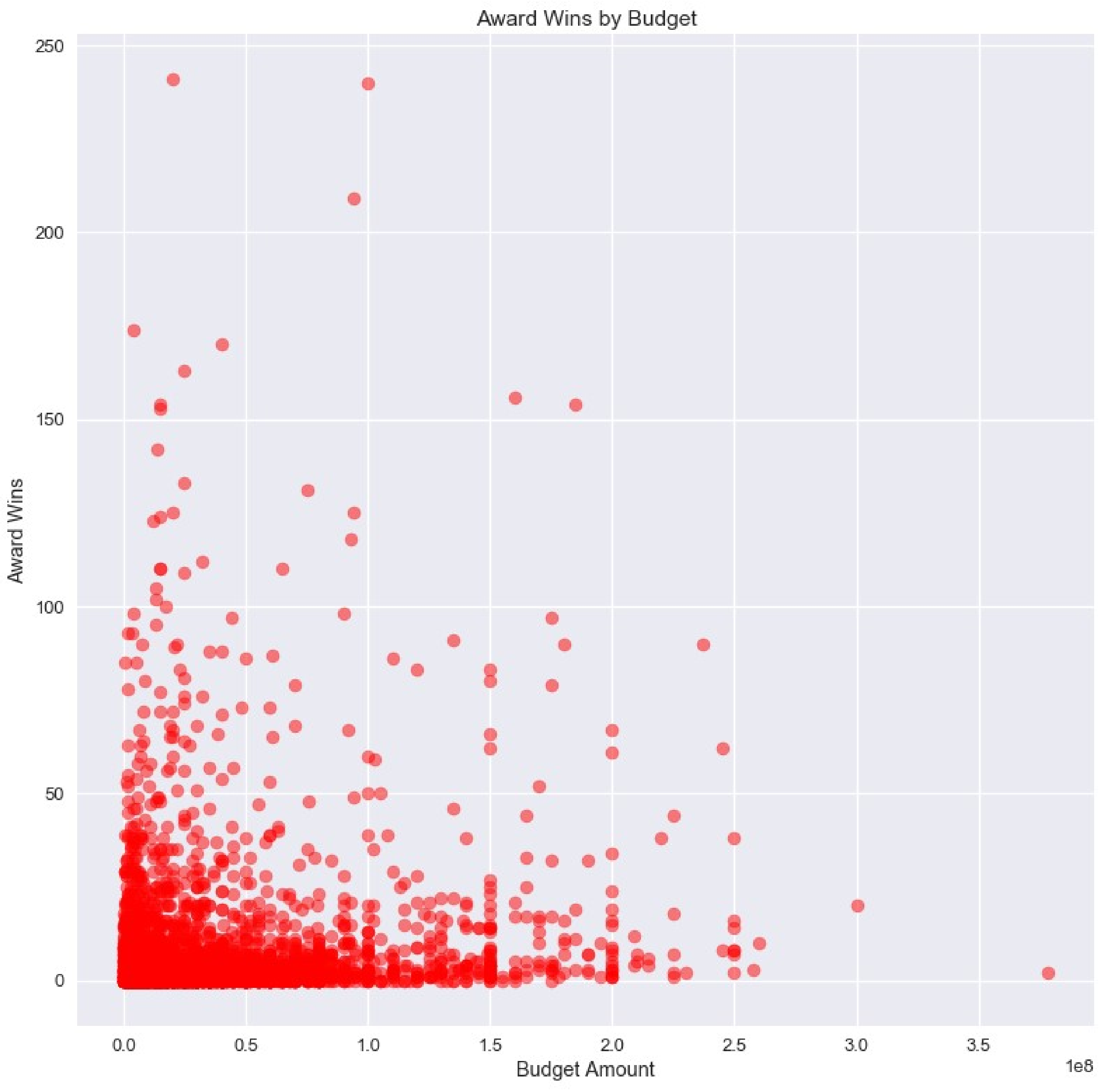
In [135]:
```
new_df.drop(['Unnamed: 0'], axis = 1, inplace = True)
new_df = new_df.sort_values(by = ['year'])
new_df.head()
```

Out[135]:

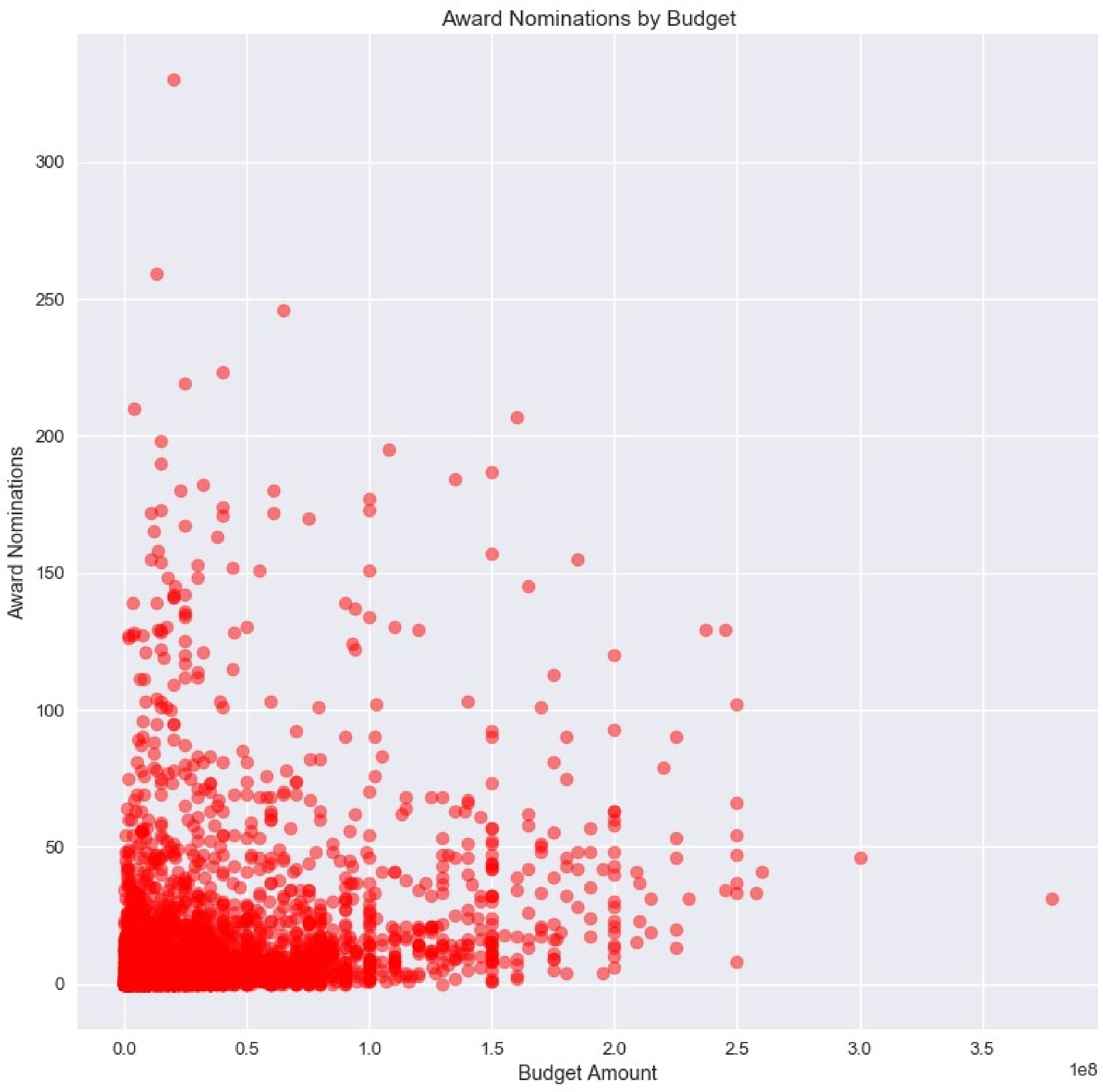
	title	year	movie_id	imdb_url	...	metacritic	rotten_tomatoes	award_nomina
3413	It Happened One Night	1934	25316	http://www.imdb.com/title/tt0025316/	...	87	98	2
3251	Modern Times	1936	27977	http://www.imdb.com/title/tt0027977/	...	96	100	1
3264	Snow White and the Seven Dwarfs	1937	29583	http://www.imdb.com/title/tt0029583/	...	95	98	6
3044	Gone with the Wind	1939	31381	http://www.imdb.com/title/tt0031381/	...	97	92	9
3236	Mr. Smith Goes to Washington	1939	31679	http://www.imdb.com/title/tt0031679/	...	73	95	12

5 rows × 10 columns

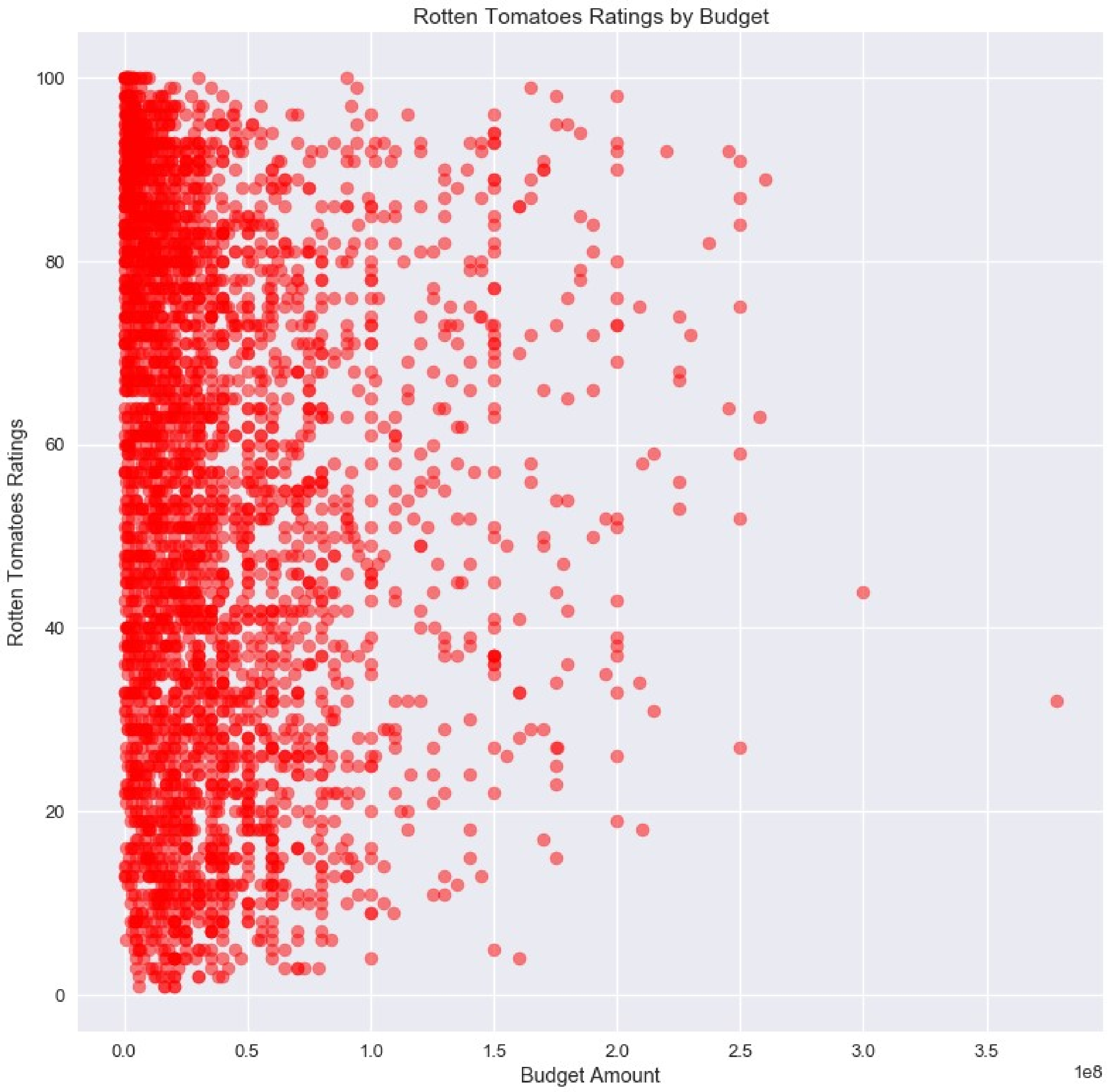
```
In [136]: #Data Visualization & Analysis
          #Award Wins by Budget

          plt.figure(figsize = (13,13))
          plt.scatter(new_df['budget_amount'], new_df['award_wins'], color= 'red', alpha =
          0.5)
          plt.title('Award Wins by Budget')
          plt.ylabel('Award Wins')
          plt.xlabel('Budget Amount')

Out[136]: Text(0.5,0,'Budget Amount')
```



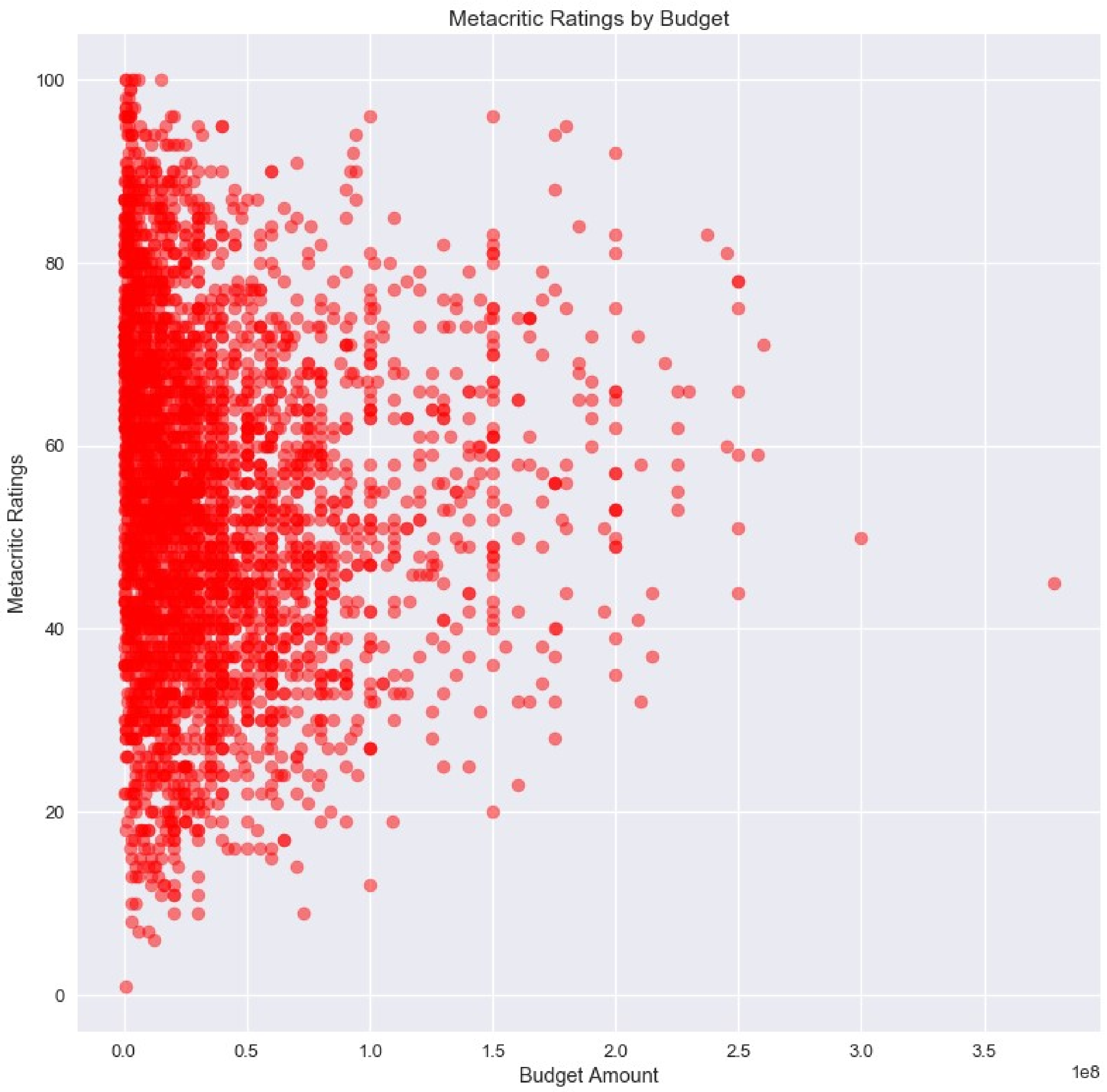
```
In [137]: #Award Nominations by Budget
          plt.figure(figsize = (13,13))
          plt.scatter(new_df['budget_amount'], new_df['award_nominations'], color= 'red', al
          pha = 0.5)
          plt.title('Award Nominations by Budget')
          plt.ylabel('Award Nominations')
          plt.xlabel('Budget Amount')

Out[137]: Text(0.5,0,'Budget Amount')
```



```
In [138]: #Rotten Tomatoes and Budget
          plt.figure(figsize = (13,13))
          plt.scatter('budget_amount', 'rotten_tomatoes', data = new_df, color = 'red', alph
          a = 0.5)
          plt.title('Rotten Tomatoes Ratings by Budget')
          plt.ylabel('Rotten Tomatoes Ratings')
          plt.xlabel('Budget Amount')

Out[138]: Text(0.5,0,'Budget Amount')
```



```
In [139]: #Metacritic Ratings and Budget
          plt.figure(figsize = (13,13))
          plt.scatter('budget_amount', 'metacritic', data = new_df, color = 'red', alpha =
          0.5)
          plt.title('Metacritic Ratings by Budget')
          plt.ylabel('Metacritic Ratings')
          plt.xlabel('Budget Amount')

Out[139]: Text(0.5,0,'Budget Amount')
```



10. Analysis Write-Up

The dataset produces exponential and poisson distributions regarding awards and ratings, respectively. In all sets, most of the data is relative, but there are a large handful of outliers to help sque the results. In the scatter plot depicting Budget Amount vs. Award Wins, you can see there is a solid drop off after the movie budget passes 1.5 million dollars. A very small set of the data lies beyond the 1.5 million mark showing that the number of nominations starts to dwindle. In the scatter plot depicting Budget Amount vs. Award Nominations, a majority of the data is confined to below one million dollars. This makes sense that movies would garner more nominations than awards, however it does show a movie's budget can only influence its award buzz to a point.

In the scatter plot depicting Budget Amount vs. Rotten Tomatoes Ratings, there are few movies with below a 40% rating with a budget of over one million dollars. There are triple the movies with a rating above 40% that have a budget over one million. However, there are more movies with a rating above 40% with a budget of less than a million than there are movies with a rating below 40% and a budget of less than one million. In the scatter plot depicting Budget Amount vs. Metacritic Ratings, the distribution of movies is centered around a 60% rating with a budget of below one million. When looking at both Budget Amount vs. Rotten Tomatoes Ratings and Budget Amount vs. Metacritic Ratings, there are a sizable amount of movies that receive higher ratings above the one million budget mark. However, the vast majority of movies with a rating over 60% lie below the one million dollar budget mark.

11. Ethics & Privacy

When considering this project, we planned to be very careful of how our analysis could affect the world. The personal information of the critics and anyone associated with the films is obfuscated so that no one will be harassed or attacked. We also protected any sensitive financial information we came across. Though we did not find anything that is meant to be kept private, it is important to consider the possibility of private data being found. On a less immediate note, it is important to keep in mind the consequences of our discussion. The artistic merit of the films is subjective and we cannot claim to ever fully represent it with an objective number. We kept our discussion open-minded so as not to condemn films that do not fit with our quality metrics and are instead valuable in other realms.

12. Conclusion & Discussion

In our pursuit to figure out if a movie's budget correlates with critical acclaim, we narrowed in the hypothesis that at a certain point, a budget does not take a movie's rating past a certain point. We took to IMDb to gather data on movie's budgets and awards activity and Rotten Tomatoes for audience and critic ratings of movies. Many movies receive award buzz and rave reviews with a very low budget, but as the budget climbed, large numbers of nominations and high ratings grew as well. However, somewhere between 1 and 1.5 million dollar budgets, the number of movies with multiple awards and high ratings starts to shrink. After gathering the data, we can conclude that a big budget will help a movie gain good reviews and generate a decent amount of awards buzz, but only to a point. After passing a 1.5 million dollar budget, the chances of a movie gaining more awards or getting higher ratings are unlikely.

Movie ratings play a vital role in the overall success of a film. Within our project, it can be said that movies should not rely solely on a large budget for the quality of their movie. A well-budgeted movie does not necessarily mean that a movie would have great quality. In our current society, blockbuster hits tend to be the movies that grab our attention, but time and time again, their large budgets disappoint us with low quality storylines. A movie can be better with a large budget, but ratings and award winnings show that the majority of successful movies do not need massive budgets. In the future, filmmaking may be addressed differently. Many films currently use budgets mainly for their CGI and design of the films, but have mediocre storylines. Our findings may help visualize that the quality of the story matters more than the designing process of the film.