Milestone 1

Varuni Gang, Andrew Greene, Scott Shaffer, Liang-Liang Zhang

April 2, 2017

API Exercises

API code to access the genre and movie poster path of your favorite movie

We used the `tmdbsimple` package as a wrapper to access TMDb.

```
import tmdbsimple as tmdb
tmdb.API_KEY = '' # Actual key not included in this report.

#  Search on 'War Games' and return TMDb movie ID and release date
#  Note: There may be multiple multiple responses to our search query
#  so the release date will help identify the correct TMDb movie id
search = tmdb.Search()
response = search.movie(query='War Games')
for s in search.results:
   print(s['title'], s['id'],  s['release_date'])

#  The movie id for War Games is '860'
wargames_tmdbmovie = tmdb.Movies(860)
response = wargames_tmdbmovie.info()

#  Obtain the genre
print wargames_tmdbmovie.genres

#  Print the poster path
response = wargames_tmdbmovie.info()
wargames_poster_path = wargames_tmdbmovie.poster_path
print wargames_tmdbmovie.poster_path
```

Genre for this movie listed by TMDb and IMDb

Note: We obtained the TMDb movie genre list for our favorite movie in the previous Python code snippet. For this part below, we used the IMDbPY package as wrapper to access the IMDb API.

We have chosen the movie *WarGames*, a 1983 cautionary tale about machine learning and relying too much on sophisticated autonomous computer models.

```
from imdb import IMDb
movieimdb = IMDb()

s_result = movieimdb.search_movie('War Games')
for item in s_result:
   print item['long imdb canonical title'], item.movieID

wargames_imdbmovie = movieimdb.get_movie('0086567')
print wargames_imdbmovie['genres']
```

The IMDb genre classification for *WarGames*: Sci-Fi, Thriller

TMDb genre classification for *WarGames*: Thriller, Science Fiction

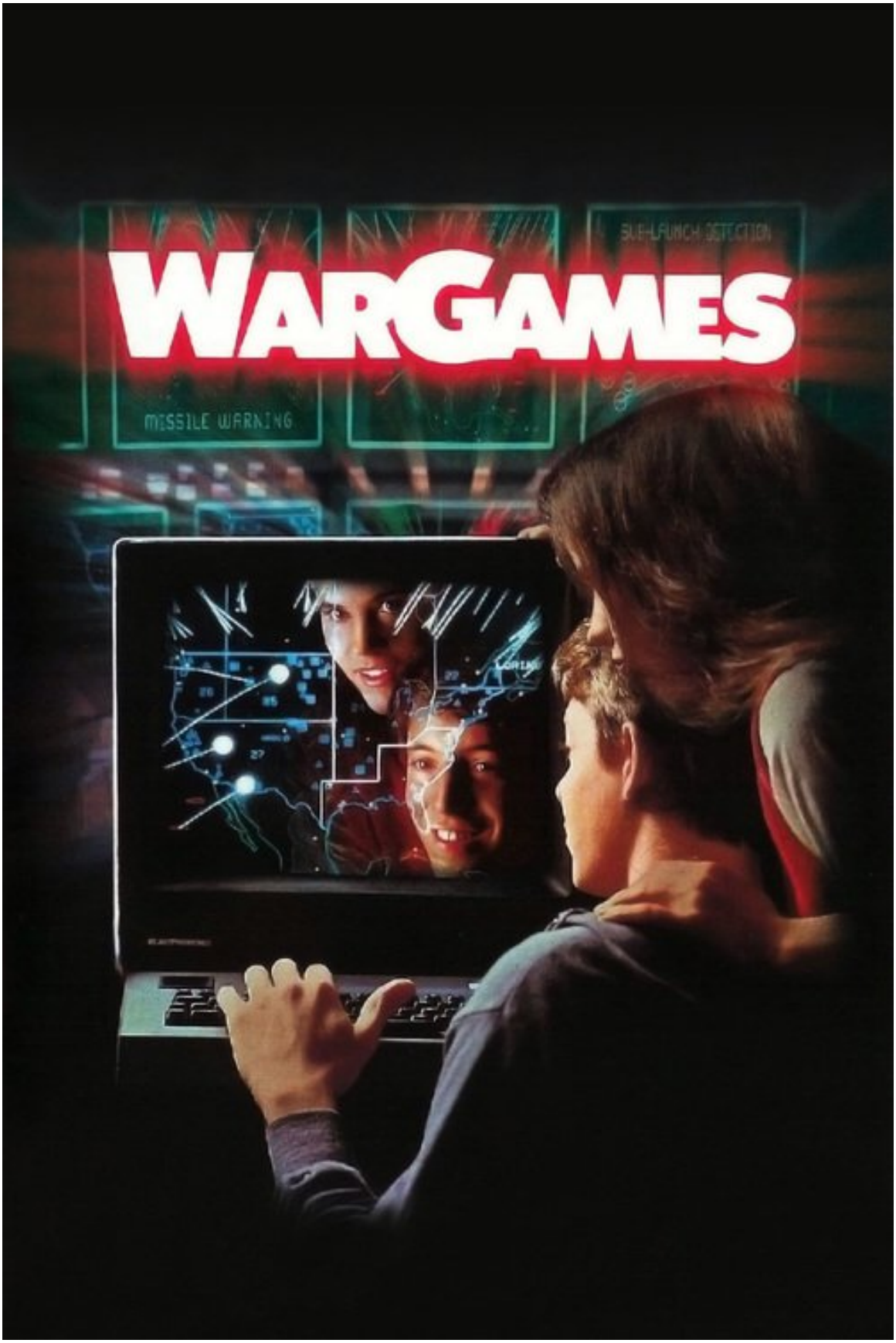


Figure 1: Movie poster image for WarGames

A list of the 10 most popular movies of 2016 from TMDb and their genre obtained via the API

```
# This code block depends on the tmdb variable, which was initialized above.
# Use TMDb discover method to obtain the 10 most popular movies of 2016
discover = tmdb.Discover()
discover_response = discover.movie(
    primary_release_year=2016,
    sort_by='popularity.desc')

print "Movie Title:Genre(s)"
for d in discover.results[0:10]:
    print d['title'] + ': ' + ', '.join([genres_by_code[gid]
                                         for gid in d['genre_ids']])
```

Observation: The 10 most popular movies of 2016 and their genres, as provided by TMDb, are the following:

Movie Title	Genre(s)
Sing	Animation, Comedy, Drama, Family, Music
Fantastic Beasts and Where to Find Them	Adventure, Action, Fantasy
Finding Dory	Adventure, Animation, Comedy, Family
Deadpool	Action, Adventure, Comedy, Romance
Rogue One: A Star Wars Story	Action, Drama, Science, Fiction, War
Arrival	Drama, Science, Fiction
Doctor Strange	Action, Adventure, Fantasy, Science, Fiction
Captain America: Civil War	Action, Science, Fiction
Underworld: Blood Wars	Action, Horror
Zootopia	Animation, Adventure, Family, Comedy

Beginning data analysis

Comment on what challenges you see for predicting movie genre based on the data you have, and how to address them

First, the genre data have some inconsistencies.

- In both TMDb and IMDb, a given movie may be tagged as having multiple genre classifications. We will discuss below our approach to this concern.
- For a given movie, the movie genre list returned by TMDb and IMDb may disagree.
- Equivalent genres may have different names between the two databases. For example, the genre that TMDb calls "Science Fiction" is known as "Sci-Fi" in the IMDb dataset. Further, some genres exist in one dataset without a corresponding genre in the other. TMDb includes "TV Movie" while IMDb includes "Biography", "Film-Noir" (presumably hyphenated to make it a single token), "Musical" (distinct from "Music"), and "Sport".

Specifically, TMDb movie genres are the following: Action, Adventure, Animation, Comedy, Crime, Documentary, Drama, Family, Fantasy, Foreign, History, Horror, Music, Mystery, Romance, Science Fiction, TV Movie, Thriller, War, Western.

Note that "Foreign" does not show up in TMDb's list of genres returned by the `genre` API call, but does show up in the actual data returned by the `movie` API call.

IMDb movie genres are the following: Action, Adventure, Animation, Biography, Comedy, Crime, Documentary, Drama, Family, Fantasy, Film-Noir, History, Horror, Music, Musical, Mystery, Romance, Sci-Fi, Sport, Thriller, War, Western.

We might consider remapping IMDb genres that have no corresponding TMDb genre into a similar genre present in TMDb; for example, "Musical" might be remapped to "Music". However, relying on the TMDb genre labels will probably suffice for this project.

There are other issues and limitations with the data as well.

TMDb's budget numbers are not always plausible; we have not yet tried using the IMDb data instead. If that is also a problem, we may try to find more reliable data (for example, from `www.the-numbers.com`, although that is a commercial site and we may not be able to use their data).

Using the actual scripts of the movies would provide a rich vein of information, but there is not a legal source for such data.

While the cast data is listed in a given order, there is not a useful distinction made between leading roles, character roles, and supernumeraries. This will make our actor-genre affinity scores less accurate than they would otherwise be.

How we are representing genres

We have decided to treat each genre label as an independent outcome. Thus, a romantic comedy would have `TRUE` for the `genre_Romance` and `genre_Comedy` columns.

This approach has several advantages.

First and second, it makes scoring our loss more fair *and* more productive. For example, if we successfully predict that the romantic comedy is a "Romance" but fail to predict that it is a "Comedy", we want our accuracy to reflect that we were half correct – and we want the feedback into our model to reflect *which* half.

Third, it eliminates the risk of creating a hand-curated list of hybrid genres that will fail to predict some new fusion film that might occur in the future.

Fourth, it allows us to use our "genre affinity" approach, as described below, to affiliate actors and directors with the genres in which they appear most often.

Code to generate the movie genre pairs and a suitable visualization of the result

Please note that this code is run after the data-acquisition code listed in the appendix.

```
# load library
library(GGally)
library(cluster)
library(ggplot2)

# read data
tmdb <- read.delim('tmdb.tsv')
tmdb_genre <- tmdb[, 12:ncol(tmdb)]
colnames(tmdb_genre) <- sub('genre_', '', colnames(tmdb_genre))

# plot correlation heatmap
library(GGally)
ggcorr(tmdb_genre, method = c("pairwise", "pearson"),
       layout.exp  = 1, hjust = 0.8, size = 3,
       legend.size = 6) +
  ggtitle('Correlation heat map for movie genre') +
  theme(plot.title = element_text(size = 12, hjust = 0.5))

# apply hierarchical clustering to reorder the genres
# so we can get clear view of how genres clustered together
```

```
tmdb_cor <- cor(tmdb_genre)
tmdb_genre.agnes = agnes(tmdb_cor, stand=T)
ggcorr(tmdb_genre[,tmdb_genre.agnes$order.lab],
       method = c("pairwise", "pearson"),
       layout.exp  = 1, hjust = 0.8, size = 3,
       legend.size = 6) +
  ggtitle('Correlation heat map for movie genre') +
  theme(plot.title = element_text(size = 12, hjust = 0.5))
```

To examine how often genres are mentioned together, a pair-wise Pearson correlation matrix was calculated for the (dummy) variables of all the move genres. In addition, hierarchical clustering was performed on the correlation matrix so that those genres with higher correlations are closer to each other by order. The resulting figure of the correlation heat map is shown below.

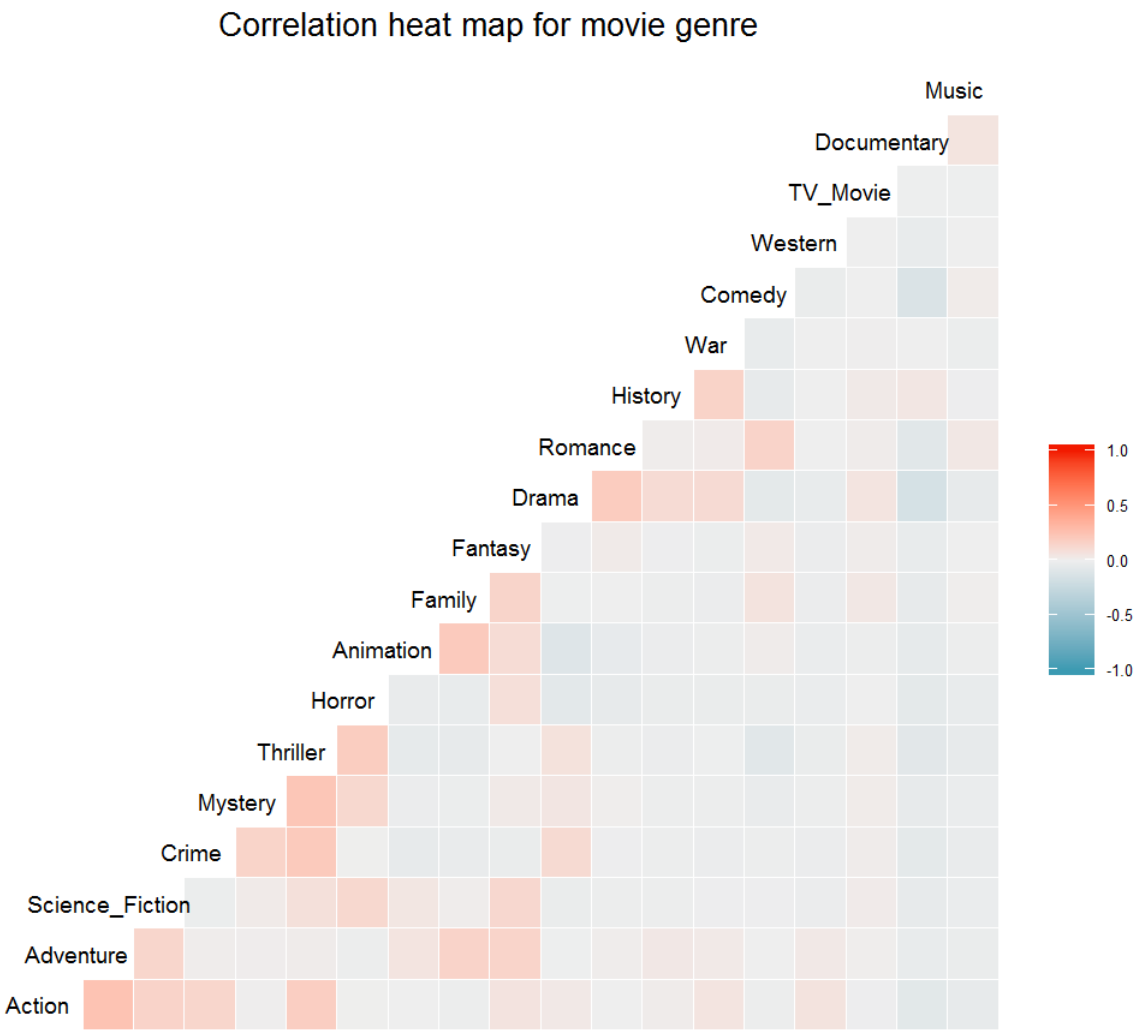


Figure 2: Correlation Heat Map

As we can see from the heat map, many genres tend to appear together. For example, the strongest correlations were observed for action/adventure/science_fiction, mystery/thriller/horror, animation/family/fantasy, drama/romance, war/history. And horror romance is definitely less likely to occur than the drama romance.

Of course, this visualization only looks at co-occurring *pairs* of genres. There are films listed with three or more genres, and those higher-order co-occurrences are omitted from this visualization.

Sketches and Exploratory Analyses

Additional visualization sketches and EDA with a focus on movie genres

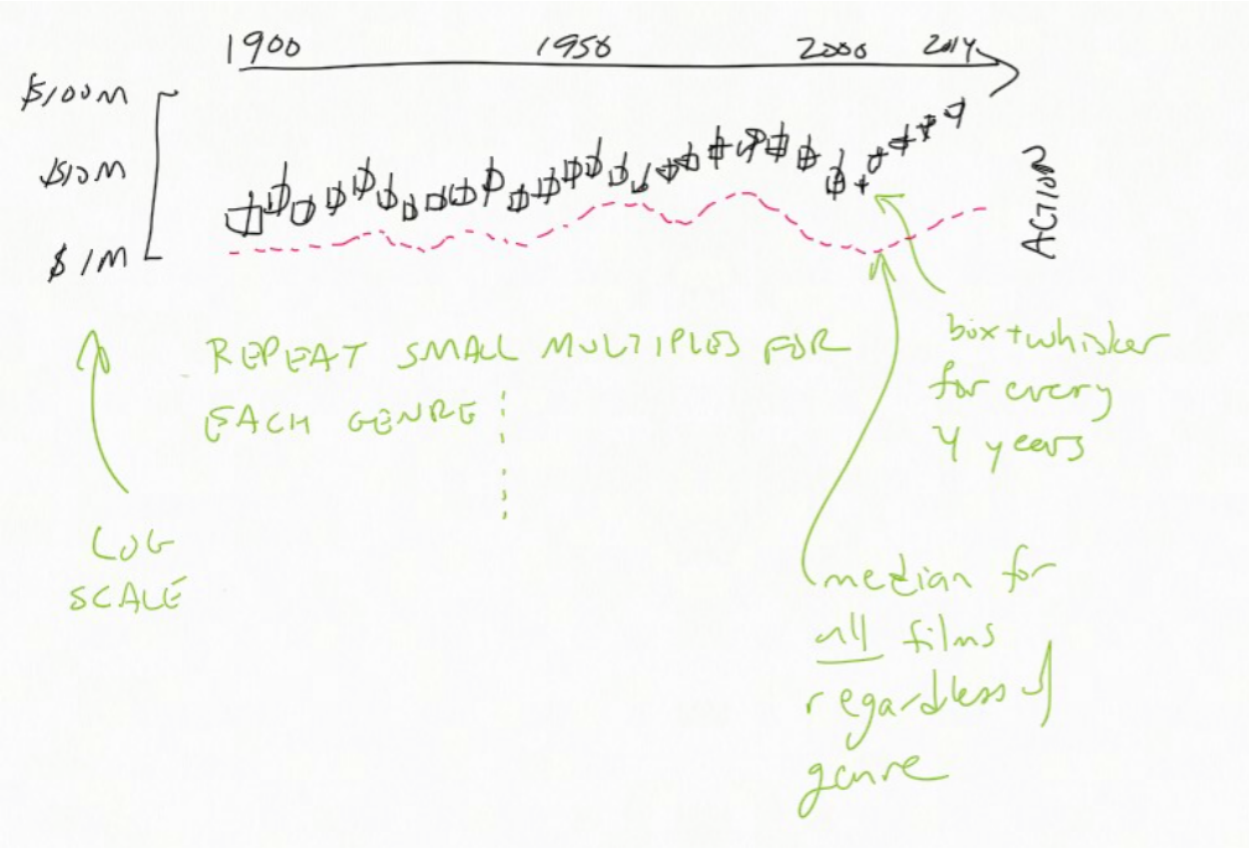


Figure 3: Visualization Sketch 1

The point of this sketch is to see if the movie's budget can be used as a predictor of genre. We first combine years into U.S. Presidential terms (for reasons we discuss below) and then create a boxplot of the budgets of in-genre movies during that time period. For comparison, a red line shows the median budget for *all* movies during that time period. We use a log scale for the y-axis because we are dealing with both large-budget and small-budget films.

When we actually generate this graph, we find some interesting things.

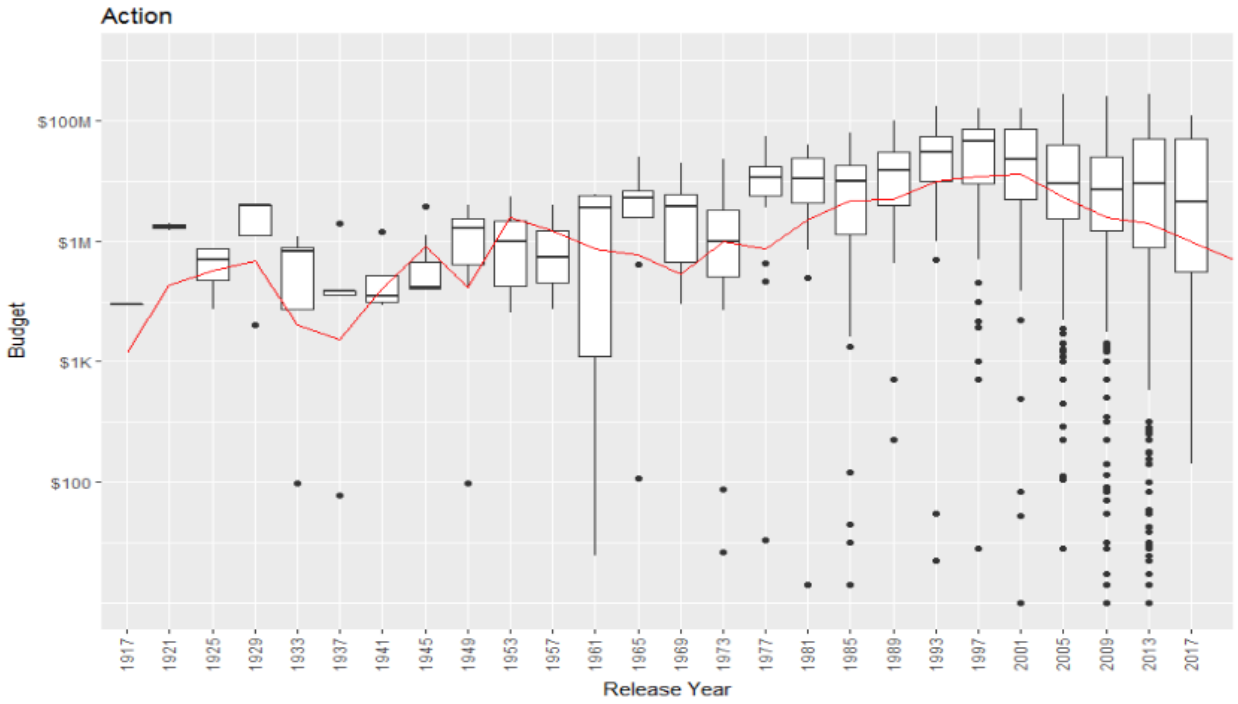


Figure 4: EDA Result 1

For action films, we expected that the budget would be significantly above the median for all films. Being misled by blockbusters, we forgot about low-budget action flicks!

Nevertheless the number of outliers with budgets in the tens and hundreds of dollars calls the accuracy of the TMDb data into question. We will need to compare these numbers with those from IMDb and decide how to deal with them.

Also, it appears from this chart that the budget information must be using inflation-adjusted dollar figures. However, it is difficult to confirm that from the API documentation and if the data are crowdsourced it may also not be consistently computed.

The code to generate this chart is given in the appendix.

A similar approach can be used to validate our approach to genre affinity

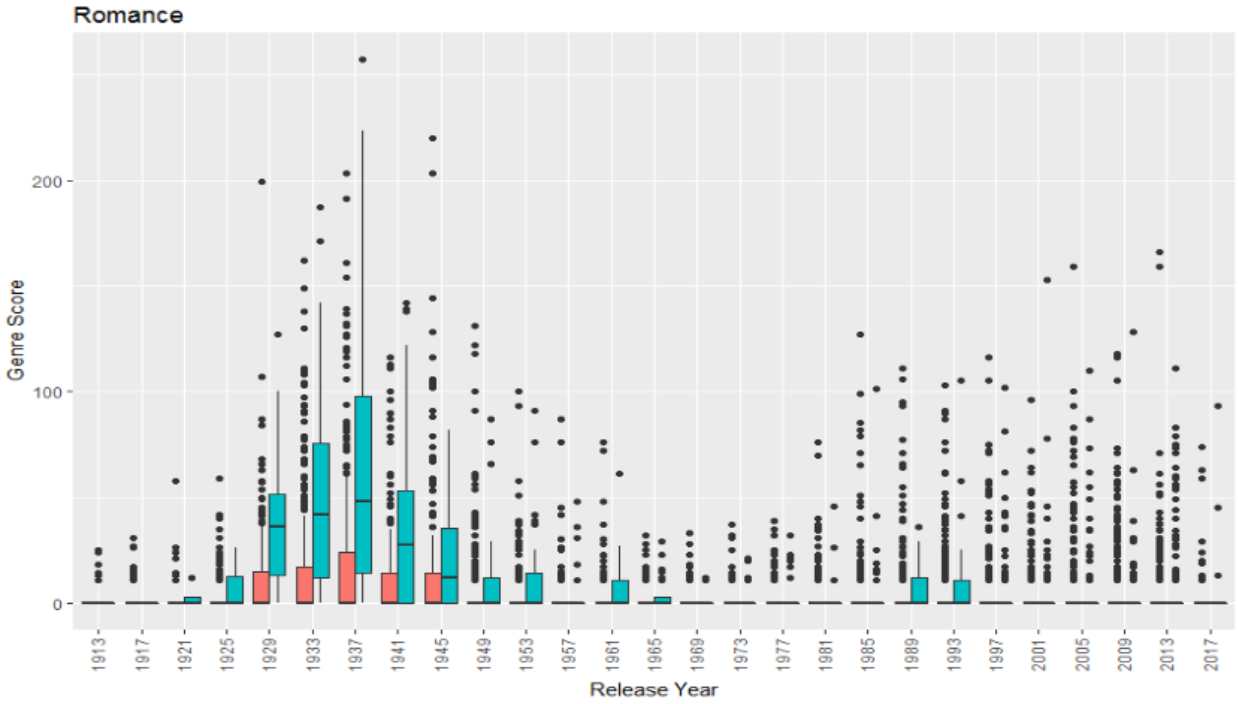


Figure 5: EDA Result 1

Here, the two set of box-and-whisker plots show the distribution of *test* data cast-genre affinity scores (computed using the *training* data to determine actor-genre affinities). Blue indicates in-genre movies and red indicates out-of-genre movies. We can see that for our sample, especially during the 1930s and 1940s, there is a strking difference between the distributions of scores of the in-sample and out-of-sample movies. In particular, the IQRs barely overlap, although the outliers have a similar range.

This suggests that our approach has some validity, but will need to be refined for it to produce a truly valuable predictor.

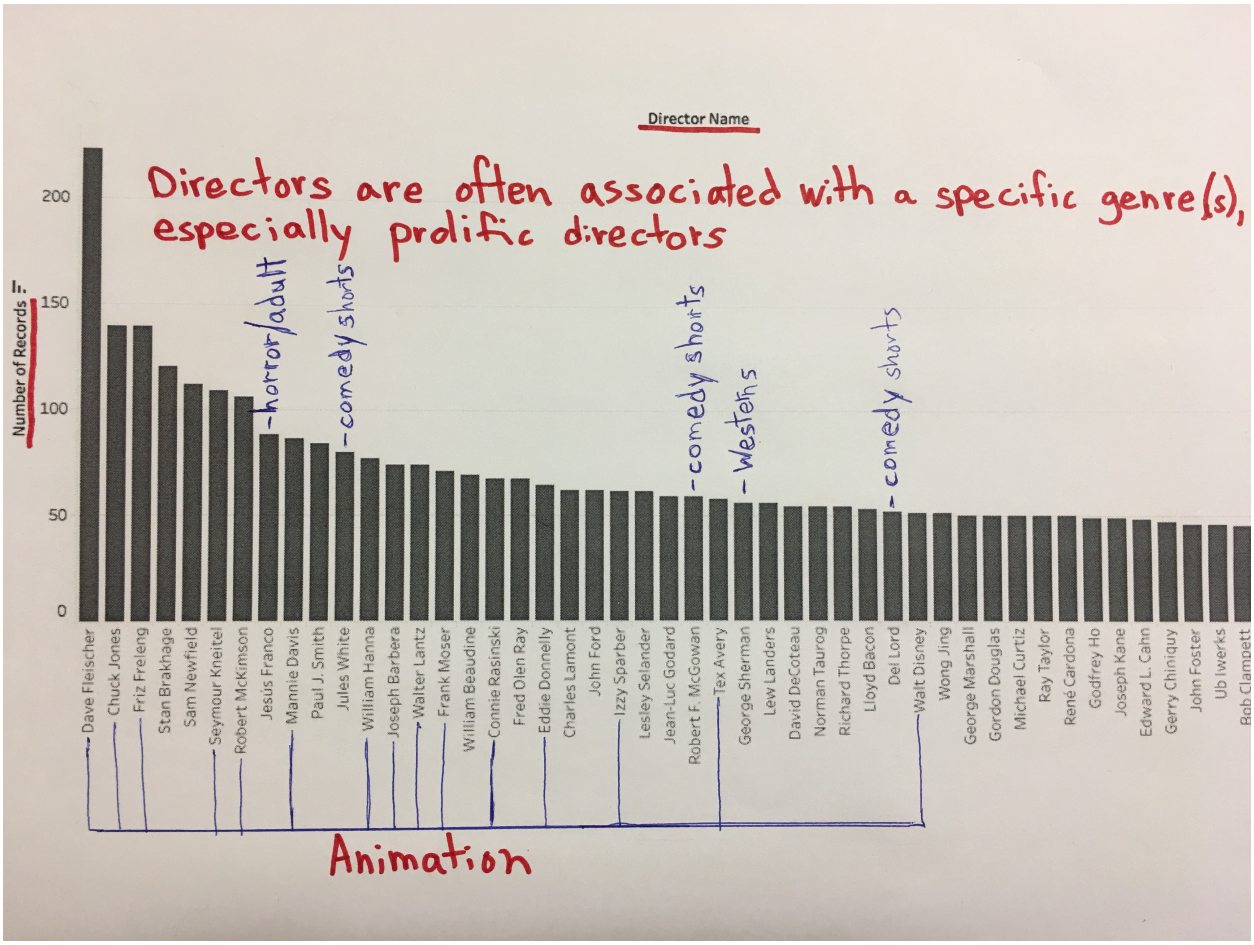


Figure 6: Visualization Sketch 2

The sketch above was created to initially explore if a movie director might be a useful predictor for genre. Here, the focus is on the most prolific directors. We performed a quick (manual) background check on the most prolific directors and many are associated with a specific genre. For example, many of the most prolific directors are almost exclusively associated with the Animation genre.

The sketch above indicates it may be worthwhile to include director as a predictor for movie genre. As part of our modeling, we may explore using Natural Language Processing (NLP) and that may include incorporating metadata, such as director, into our NLP model.

Another way of examining this is by assigning "director-genre affinities".

In this plot, each movie is a single dot; again blue means "in-genre" (in this case, "Romance") and red means "out-of-genre". The x-axis is the year of release and the y-axis is the director-genre affinity score for the movie. Red dots on the x-axis are true negatives, blue dots above the axis are true positives. Red dots above the axis are false positives – each is an out-of-genre film whose director has nevertheless made that many in-genre films. Similarly, a blue dot on the x-axis is a false negative – an in-genre film made by a director whose in-genre output is otherwise meagre.

- *A list of questions you could answer with this and related data. Get creative here!*

The `release_date` can be broken out in three *categorical* predictors: one for the month of release (e.g. to capture sunmmer action films), one for the year mod 4, and one for the year converted into a presidential term (i.e., `4*floor((year-1)/4)+1`). The latter two are because there is a 4-year economic cycle, and because we want to capture the idea that different genres may have been popular during different times – but not in a

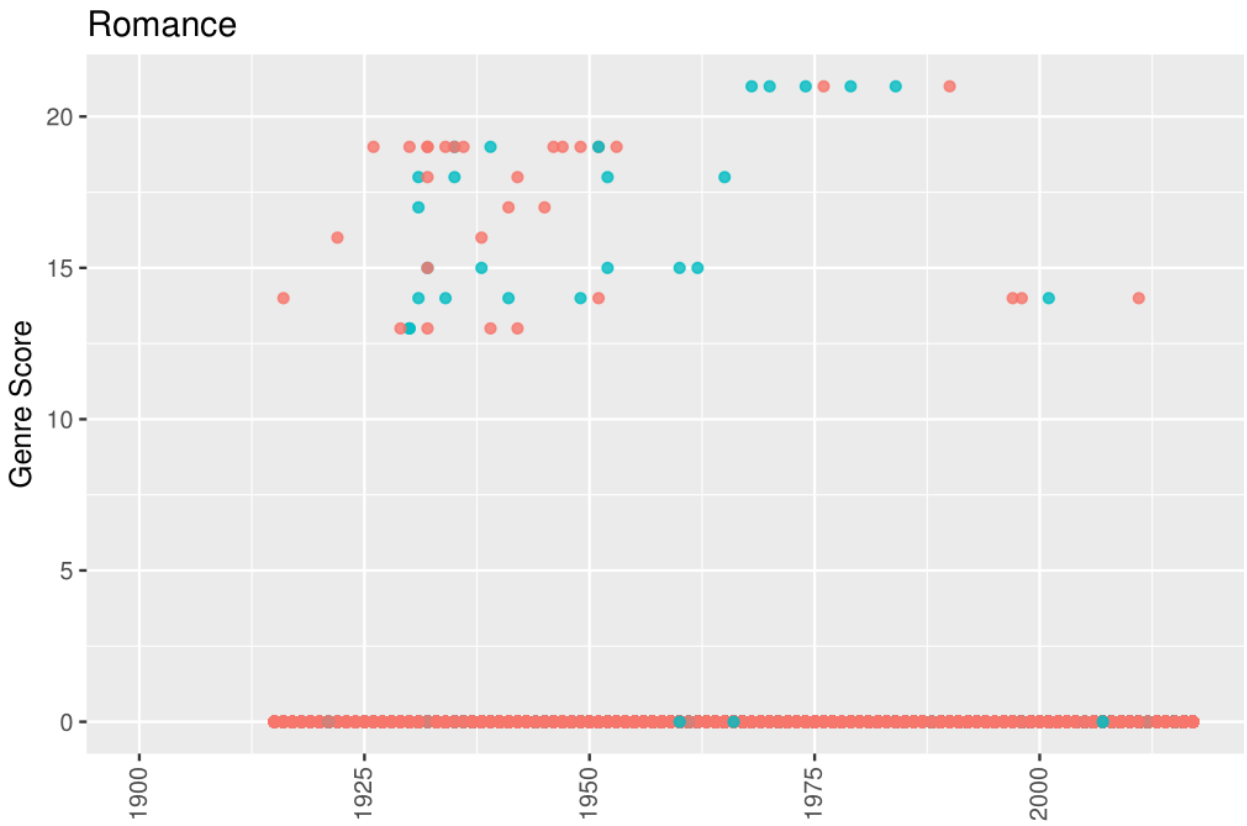


Figure 7: EDA Result 1

way that is modeled by "time moves forward." Because so many cultural and economic developments in the US are related to the presidential terms, this seems like an approach that might bear fruit.

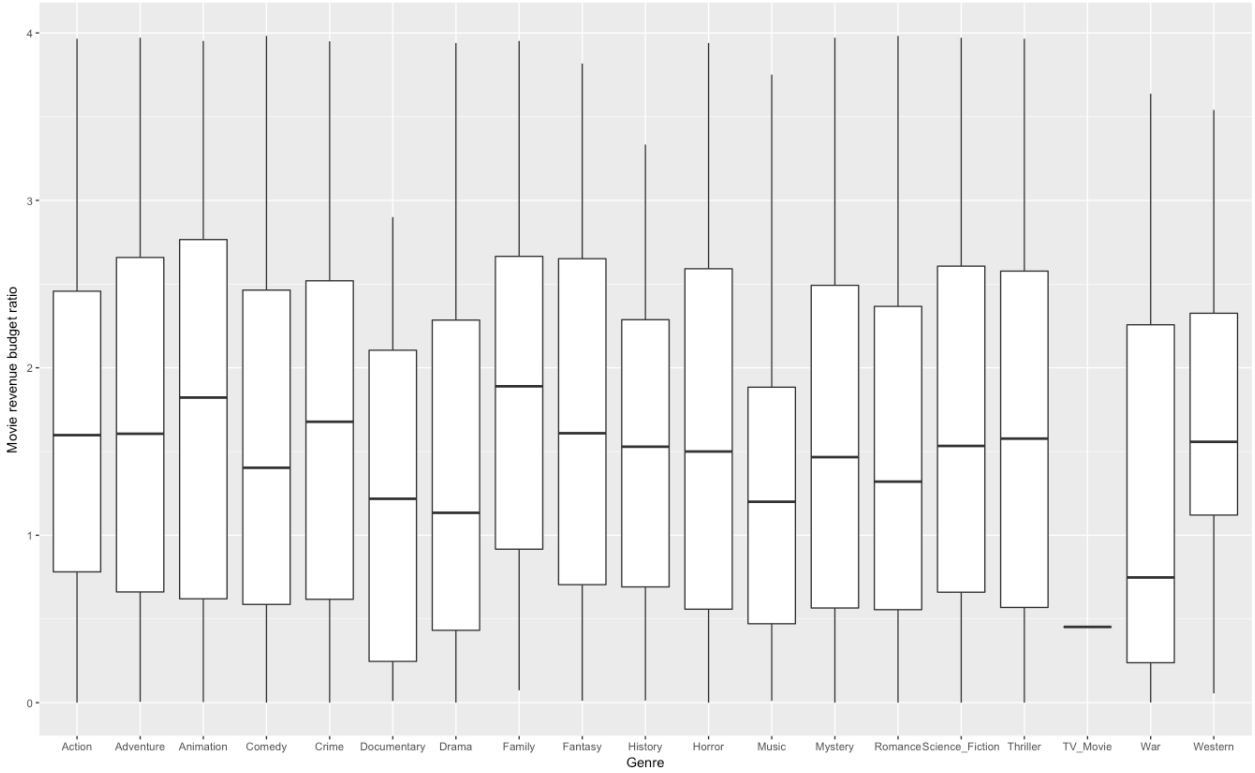
Would it be productive to consider a different class of model for each genre? Since we have decided to represent each genre as a separate binary outcome, we don't have a multi-class classification problem and there is no particular reason to force them into a common template.

We have not yet looked at bag-of-words or other language processing on the title or description. Assuming that we include the genre names themselves on our list of stopwords (to do otherwise would be like cheating), how useful would those predictors be?

Do we see a correlation between popularity of movies and particular lead actor(s) across genres? For example, we may expect Seth Rogan to lead in popular comedies, but lead in relatively unpopular dramas.

Do we see any patterns of actors working together in particular genres? An example would be Emma Stone and Ryan Goseling in Romance/Comedy.

It may be worth investigating whether there is a correlation between budget and genre for popular movies. Do we see some genres that have higher budgets than others, and do we see this more pronounced in the popular subgroup? In addition, do we see any pattern in revenue:budget ratios - are any genres more "bang for the buck" than others? From our preliminary visualization, we can see that TV_movies appear to be extremely inefficient in terms of spending and profit, although their business model is not "selling tickets" but rather "selling ad time" and so the way revenue is accounted for may differ. Documentary and Music seem to underperform the others, and War has a very unusual profile (very low median but a large third quartile). It remains to be seen whether these distributions will recommend this measure as a predictor for certain genres.



As mentioned in the assignment, it would be worth asking whether there is a relationship between genre and the length of the movie title. Below we have included our preliminary visualization based on a small subset of the data that we have. Some genres appear to be atypical: Documentary, Music, and TV Movie. There is more to be hashed out here.

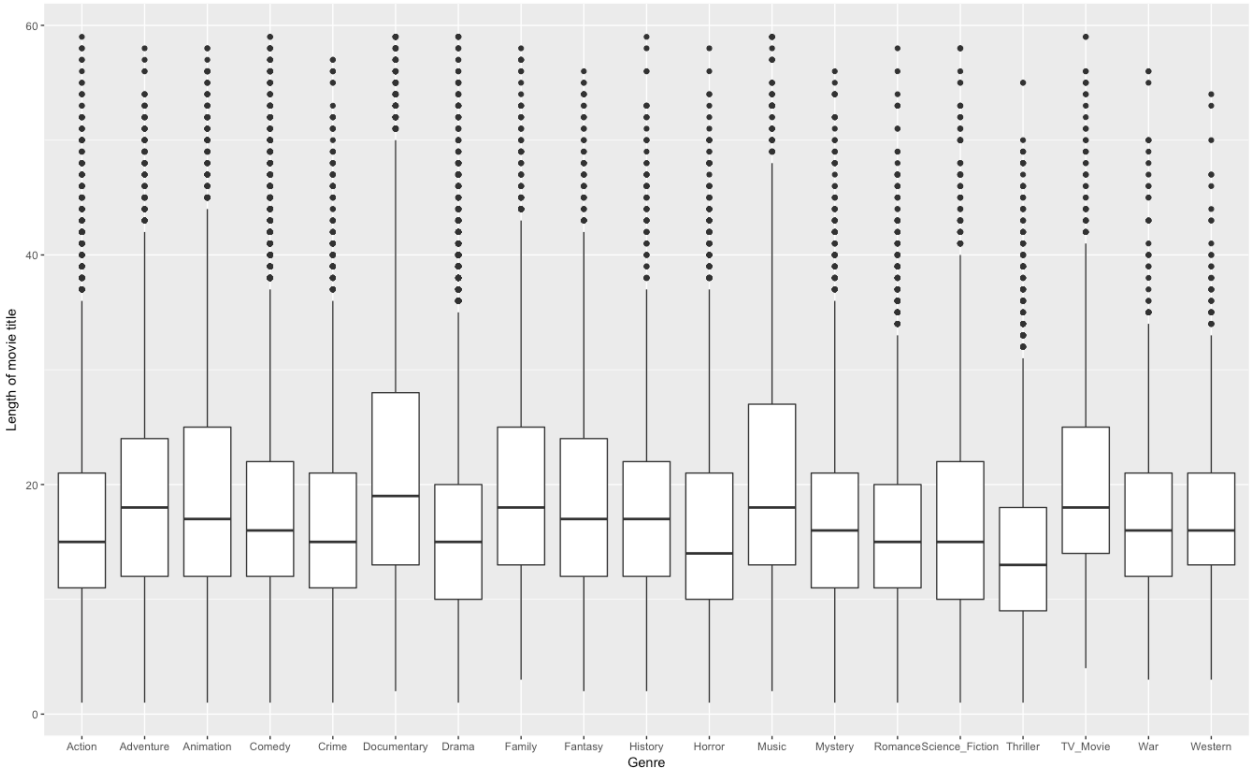


Figure 8: Length of Movie Title

Do certain genres fall within a range of predictable run-times? We may anticipate that animation films would have shorter run-times than dramas, and comedies may have shorter run-times than thrillers due to plot development, budget. . . etc. This preliminary visualization provides an exciting finding. Animation films very clearly have a shorter run-time than any other genre. As some may have expected, history films appear to drag on.

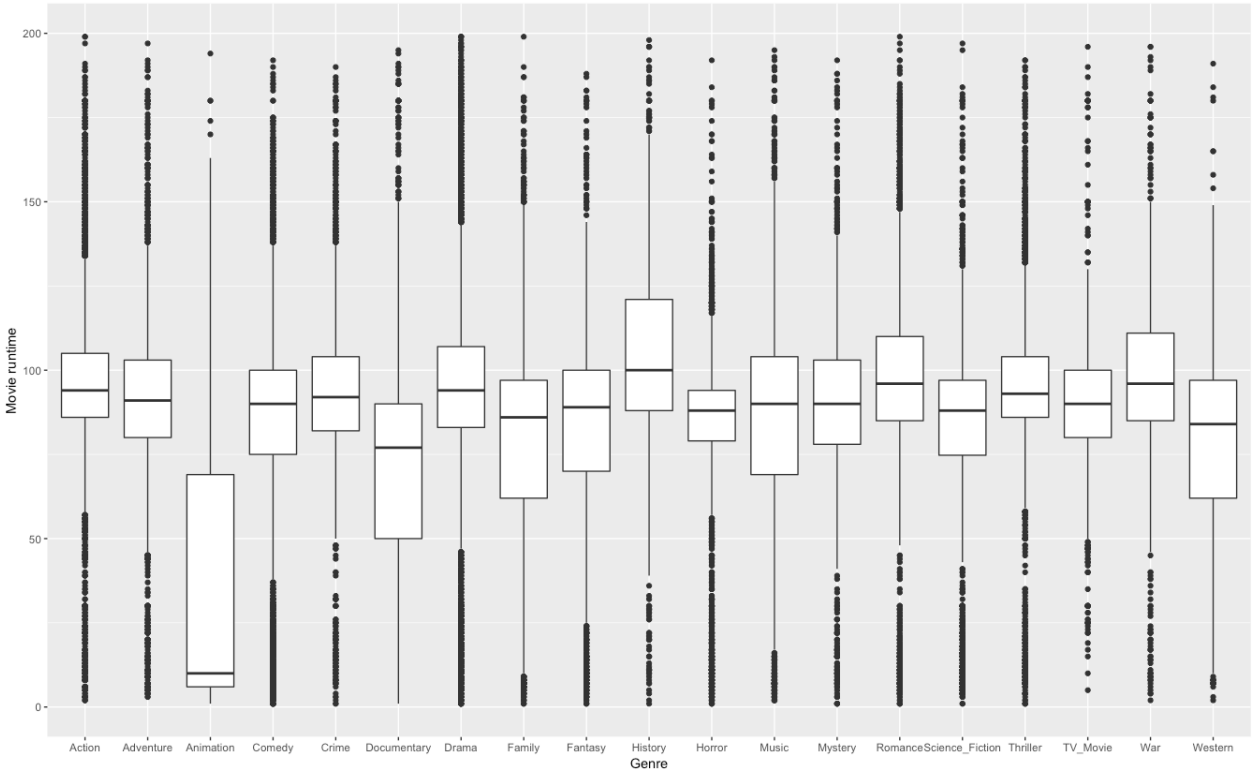


Figure 9: Movie Run-time

Our heat map scratches the surface in regards to one genre's ability to predict another for a given movie. We have more work to do in terms of teasing out the appearance of multiple genres for one film. Another interesting question may be whether any genres are absolutely not found together (*e.g.*, war and family films) and whether these correlations can be used to augment the models we build.

As mentioned previously, certain language is used more often in certain genre contexts. We may see this play out in movie titles. For example, words like "war" and "blood" may have an affinity to action and thriller movies.

Do we see a trend in popularity around the time of the Academy Awards, and what can this tell us about genre patterns around that release period?

Additional material

Here is how we have downloaded and pre-processed our data. All code and generated `.tsv` data files are available at our git repo, `https://github.com/amgreene/cs109b`

We begin by using the `discover` method in the TMDB API to enumerate all movies in the database:

```
''' Initial TMDB step is to use "discover" to find out what movies they have
'''
```

```
from json import loads
from time import sleep
from urllib2 import urlopen

api_key = open('apikey.txt').readline().strip()
for year in xrange(2017, 1911, -1):
    # Write each year to its own file
    o = open('data/tmdb-' + str(year) + '.txt', 'w')
    page = 1
    # Each response contains a single json "page" object
    # with multiple movies
    # Each file will contain one json object per line
    # We'll parse them later
    # (Except that we need to do minimal parsing here
    # to know when we've reached the last page for the
    # given year)
    while True:
        u = urlopen('https://api.themoviedb.org/3/discover/movie?' +
                    'year=%(year)d&page=%(page)d&api_key=%(api_key)s' %
                {
                    'api_key': api_key,
                    'year': year,
                    'page': page,
                   })
        # If it excepts, we'll let the script die and recover manually
        response = u.read()
        o.write(response + "\n")
        max_page = loads(response)['total_pages']
        print year, ":", page, "/", max_page, "   \r",
        if page == max_page:
            print
            break
        page += 1
        # 250 msec = 4 request/sec = 40 req/10 sec is their rate limit
        sleep(.250)
```

We then use the `movie` method to get the detailed information, including the `IMDB_id` and the `poster_path`. Unfortunately, we didn't realize until this was almost done that we could have combined it with the call to retrieve the credits information.

```
''' Get the detailed records for each movie from TMDB
'''

from time import sleep
from json import loads
from urllib2 import urlopen, HTTPError

api_key = open('apikey.txt').readline().strip()

URL_TEMPLATE = ('https://api.themoviedb.org/3/movie/' +
                '%(movie_id)d?api_key=%(api_key)s')

for year in xrange(2017, 1914, -1):
    # One output file per year, containing each JSON
```

```
    # response object on its own line.
    o = open('data/tmdb-details-' + str(year) + '.txt', 'w')

    # Read in that year's cached responses to the
    # discover query, and iterate over them:
    for page_str in open('data/tmdb-' + str(year) + '.txt', 'r'):
        # Convert the json object with one page
        # of 'discover' responses into a dictionary
        page = loads(page_str)

        # And iterate over its 'results' element,
        # which contains a list of movies:
        for movie in page['results']:
            # Print some status so we can track progress
            # while this is running
            print ('\r', year, ":",
                   page['page'], "/", page['total_pages'],
                   'Movieid:', movie['id']),
            try:
                # Fetch the detail record for this movie
                u = urlopen(URL_TEMPLATE %
                            {
                                'api_key': api_key,
                                'movie_id': movie['id'],
                            })
                response = u.read()
                o.write(response + "\n")
            except HTTPError as e:
                print "\nEXCEPTION!"
                print e.code
                print e.reason
                # If we hit the occasional 404 error,
                # we want to ignore and keep going.
                # There's one exception: if we have
                # a bug and bust the rate limit, we
                # need to be responsible citizens and
                # bail immediately!
                if e.code == 429:
                    exit()

            # 250 msec = 4 request/sec = 40 req/10 sec
            # that's TMDB's rate limit
            sleep(.250)
```

We convert the details information into a tab-separated-values (`.tsv`) file which can be used in R or Python. As we add more fields, we'll update this script. For example, we plan to incorporate information about the cast, the director, and the producers.

```
''' Take the detalied info derived from TMDB and extract a single .tsv file
'''
import codecs
from json import loads

# The following list of genres is manually retreived
```

```
# from https://api.themoviedb.org/3/genre/movie/list?
# and then cleaned up to fit in 80 columns
genre_ids = [{"id":28,"name":"Action"},
             {"id":12,"name":"Adventure"},
             {"id":16,"name":"Animation"},
             {"id":35,"name":"Comedy"},
             {"id":80,"name":"Crime"},
             {"id":99,"name":"Documentary"},
             {"id":18,"name":"Drama"},
             {"id":10751,"name":"Family"},
             {"id":14,"name":"Fantasy"},
             # Foreign is not in that list but
             # can be deduced from the detailed data
             {"id":10679,"name":"Foreign"},
             {"id":36,"name":"History"},
             {"id":27,"name":"Horror"},
             {"id":10402,"name":"Music"},
             {"id":9648,"name":"Mystery"},
             {"id":10749,"name":"Romance"},
             {"id":878,"name":"Science Fiction"},
             {"id":10770,"name":"TV Movie"},
             {"id":53,"name":"Thriller"},
             {"id":10752,"name":"War"},
             {"id":37,"name":"Western"}]

# Which fields will we want to extract?
fields = ['tmdb_id', 'imdb_id', 'title',
          'release_date', 'budget', 'original_language',
          'popularity', 'vote_average', 'vote_count',
          'runtime', 'revenue']

# The genres will be written as a collection of boolean columns

# TODO: production_companies as factors?


# Train vs. Test:

# We will divide our data by taking the last digit of the movie's id
# If it's 0-4 then it's training data from the get-go
# 5 is test data for MS1, and training data after that
# 6 is test data for MS2, and training data after that
# 7 is test data for MS3, and training data after that
# 8 is test data for MS4, and training data after that
# 9 is test data for the final deliverable
test_digit = 5

# Open the output files and write the header row
o_train = codecs.open('data/tmdb.tsv', 'w', 'utf-8')
o_train.write('\t'.join(fields) +
        '\t' +
        '\t'.join(['genre_' + g['name'].replace(' ', '_')
                   for g in genre_ids]) +
```

```
        '\n')

o_test = codecs.open('data/tmdb-test.tsv', 'w', 'utf-8')
o_test.write('\t'.join(fields) +
        '\t' +
        '\t'.join(['genre_' + g['name'].replace(' ', '_')
                   for g in genre_ids]) +
        '\n')


# We'll want to know how many movies we've seen
# (just to report progress)
num_seen = 0

# Some films appear in more than one year's data set,
# so we need to dedup them
ids_dedup = set()

# Because we're running this while the downloads are still happening,
# we expect an "incomplete input" ValueError exception to be raised
# when we hit the end of the input file (which has not yet been
# flushed to disk). So we'll use a try/finally so we can keep track
# of how many movies we've seen so far.
try:
    for year in xrange(2017, 1914, -1):
        for movie_str in open('data/tmdb-details-' +
                              str(year) + '.txt', 'r'):
            movie = loads(movie_str)

            # Do the dedup
            if movie['id'] in ids_dedup:
                continue
            ids_dedup.add(movie['id'])

            num_seen += 1

            # If this movie's data is supposed to
            # stay concealed until a later milestone,
            # then ignore it for now.
            movie_id_last_digit = movie['id'] % 10
            if movie_id_last_digit > test_digit:
                continue

            # We want the output column to be called
            # tmdb_id rather than id so it's unambiguous
            # Simplest is to copy the existing column
            # to the name we want, and pretend that's what
            # it was called all along
            movie['tmdb_id'] = movie['id']

            # Convert the list of genres to a set which
            # can later be output as a list of booleans
            # in a consistent order
```

```
            movie_genres = set()
            for g in movie['genres']:
                movie_genres.add(g['name'])

            # Choose which output file to use based on
            # whether this is training or test data
            o = o_train
            if movie_id_last_digit == test_digit:
                o = o_test

            # Output this movie's row: First, the list
            # of fields that just get passed through,
            # and then the list of genre booleans.
            o.write('\t'.join([unicode(movie[f])
                               for f in fields]
                              +
                              ['1'
                               if g['name'] in movie_genres
                               else '0'
                               for g in genre_ids]) +
                    '\n')

finally:
    print "Done! Summarized", num_seen, "movies"
```

We have decided to model the genres as a family of binary outcomes; these are not dummy variables because a given movie may be a member of multiple genres. This gives us the ability to score each genre independently – if a movie is both a "romance" and a "comedy" and our "romance" model predicts `1` while our "comedy" model predicts `0`, that is more useful information than if we had a "romcom" model that predicted `0`.

Our current plan, subject to change, is to have separate `.tsv` files for each data source, and rely on joining them by key inside R or Python. This will give us maximum flexibility in reshaping the data from each source independently.

We are separating our data as follows: We take the last digit of the movie's TMDB id. (We have verified that this seems to be evenly spread across years and genres.)

- If the last digit is 0-4, then this movie is training data from the get-go
- If it is 5, then this movie is test data for Milestone 1, and is added to the training data after that
- If it is 6, then this movie is test data for Milestone 2, and is added to the training data after that
- If it is 7, then this movie is test data for Milestone 3, and is added to the training data after that
- If it is 8, then this movie is test data for Milestone 4, and is added to the training data after that
- If it is 9, then this movie is test data for Milestone 5 (the final deliverable)

We will allow ourselves to "peek" ahead only to download related data (e.g., to make sure we have everything from IMDB early in the process).

For each of the movies that we got from the TMDB API we used the IMDB ID that they provided to query IMDB.

```
import findspark
findspark.init()
import pyspark
sc = pyspark.SparkContext()

tmbdRDD = sc.textFile("./data/tmdb.tsv")
import re
```

```
imdb_ids = (tmbdRDD.map(lambda lines: lines.split()[1])
         .filter(lambda word: (word != 'imdb_id') & (word.startswith("tt")))
         .map(lambda word: re.sub("\D", "", word) )
         .collect())
```

We use the IDs to query IMDB for movie summaries via IMDbPY (which is a Python interface available to access the metadata) and save it into a file.

```
from imdb import IMDb
ia = IMDb()
imdbMoviefile = open('./data/imdb_movie_summary.txt', 'w')
for i in imdb_ids:
    movie = ia.get_movie(str(i))
    print imdbMoviefile.write("%s\n" % (['Id: ' + str(i)] + str(movie.summary()).splitlines()[2:]))
```

Once we have a significant chunk of movie information, we will convert the information provided in the summaries (i.e. Cast, Country, Rating, Plot…etc.) into a meaningful format. (As mentioned above, each data source will have one or more independent `.tsv` files.)

Returning to our TMDB data:

We download the credits data using a script similar to the one for the detailed information about each movie:

```
''' Get the detailed credits for each movie from TMDB
'''

from time import sleep
from json import loads
from urllib2 import urlopen, HTTPError

api_key = open('apikey.txt').readline().strip()

URL_TEMPLATE = ('https://api.themoviedb.org/3/movie/' +
                '%(movie_id)d/credits?api_key=%(api_key)s')

for year in xrange(2017, 1914, -1):
    # One output file per year, containing each JSON
    # response object on its own line.
    o = open('data/tmdb-credits-' + str(year) + '.txt', 'w')

    # Read in that year's cached responses to the
    # discover query, and iterate over them:
    for page_str in open('data/tmdb-' + str(year) + '.txt', 'r'):
        # Convert the json object with one page
        # of 'discover' responses into a dictionary
        page = loads(page_str)

        # And iterate over its 'results' element,
        # which contains a list of movies:
        for movie in page['results']:
            # Print some status so we can track progress
            # while this is running
            print ('\r', year, ":",
                   page['page'], "/", page['total_pages'],
                   'Movieid:', movie['id']),
            try:
```

```
            # Fetch the credits record for this movie
            u = urlopen(URL_TEMPLATE %
                        {
                            'api_key': api_key,
                            'movie_id': movie['id'],
                        })
            response = u.read()
            o.write(response + "\n")
        except HTTPError as e:
            print "\nEXCEPTION!"
            print e.code
            print e.reason
            # If we hit the occasional 404 error,
            # we want to ignore and keep going.
            # There's one exception: if we have
            # a bug and bust the rate limit, we
            # need to be responsible citizens and
            # bail immediately!
            if e.code == 429:
                exit()

        # 250 msec = 4 request/sec = 40 req/10 sec
        # that's TMDB's rate limit
        sleep(.250)
```

And then we convert that into a more compact `.tsv` file as well:

```
''' Take the credits data and convert it to tsvs for cast and director
'''
import codecs
from collections import defaultdict
from json import loads

# Cast
cast_train = defaultdict(list)
cast_test = defaultdict(list)
cast_names = {}

# Some films appear in more than one year's data set, so we need to dedup them
ids_dedup = set()

# We will divide our data by taking the last digit of the movie's id
# If it's 0-4 then it's training data from the get-go
# 5 is test data for MS1, and training data after that
# 6 is test data for MS2, and training data after that
# 7 is test data for MS3, and training data after that
# 8 is test data for MS4, and training data after that
# 9 is test data for the final deliverable
test_digit = 5

o_director_train = codecs.open('data/director.tsv', 'w', 'utf-8')
o_director_train.write('Movie_ID\tDirector_ID\tDirector_Name\n')
o_director_test = codecs.open('data/director-test.tsv', 'w', 'utf-8')
o_director_test.write('Movie_ID\tDirector_ID\tDirector_Name\n')
```

```
for year in xrange(2017, 1914, -1):
    print year,
    for movie_str in open('data/tmdb-credits-' + str(year) + '.txt', 'r'):
        movie = loads(movie_str)

        movie_id = movie['id']

        # Do the dedup
        if movie_id in ids_dedup:
            continue
        ids_dedup.add(movie_id)

        movie_id_last_digit = movie_id % 10
        if movie_id_last_digit > test_digit:
            continue
        cast = cast_train
        o_director = o_director_train
        if movie_id_last_digit == test_digit:
            cast = cast_test
            o_director = o_director_test

        cast_dedup = set()
        for cast_member in movie['cast']:
            # Ignore when one actor has multiple roles in one movie
            # e.g. Sellars in Strangelove
            if cast_member['id'] in cast_dedup:
                continue
            cast_dedup.add(cast_member['id'])
            cast[cast_member['id']].append(movie['id'])
            cast_names[cast_member['id']] = cast_member['name'].strip().replace('\t', ' ')

        for crew_member in movie['crew']:
            if crew_member['job'] == 'Director':
                o_director.write(unicode(movie['id']) + '\t' +
                                 unicode(crew_member['id']) + '\t' +
                                 crew_member['name'].strip() + '\n')

cast_tuples = sorted([(len(movie_ids),
                       cast_names[cast_id],
                       cast_id,
                       movie_ids)
                      for (cast_id, movie_ids)
                      in cast_train.items()])

o = codecs.open('data/credits-by-name.txt', 'w', 'utf-8')
o.write('Num_roles\tName\tCast_id\tMovie_id_list\n')
for (len_roles, cast_name, cast_id, movie_ids) in cast_tuples:
    o.write('%(len_roles)d\t%(cast_name)s\t%(cast_id)d\t%(movie_ids)s\n' % {
        'len_roles': len_roles,
        'cast_name': cast_name,
        'cast_id': cast_id,
        'movie_ids': ','.join([str(i) for i in movie_ids])
        })
```

```
cast_tuples = sorted([(len(movie_ids),
                       cast_names[cast_id],
                       cast_id,
                       movie_ids)
                      for (cast_id, movie_ids)
                      in cast_test.items()])

o = codecs.open('data/credits-by-name-test.txt', 'w', 'utf-8')
o.write('Num_roles\tName\tCast_id\tMovie_id_list\n')
for (len_roles, cast_name, cast_id, movie_ids) in cast_tuples:
    o.write('%(len_roles)d\t%(cast_name)s\t%(cast_id)d\t%(movie_ids)s\n' % {
        'len_roles': len_roles,
        'cast_name': cast_name,
        'cast_id': cast_id,
        'movie_ids': ','.join([str(i) for i in movie_ids])
        })
```

We compute actor-genre affinity scores, and then use those to compute the movie's cast-genre affinity scores:

```
''' Process the acting credits to determine actor-genre affinities
    Then use those to compute a per-movie cast-genre affinity score

    (1) For each actor, for each genre, count how many of that actor's
        films are tagged with that genre. We call that the
        actor-genre affinity score
    (2) Discard any actor-genre affinity score that is less than
        a threshold (for now, we're using 10) because that actor
        has not appeared in enough films for it to count as a
        true affinity
    (3) For each movie, for each genre, add up the (surviving)
        actor-genre affinity scores for all actors in that film.
        We call that the movie's cast-genre-affinity score.

    Things we plan to try in future versions:

    * Duplicate this for directors
    * Experiment with different threshold values
    * Disregard appearances whose position in the credits is
      beyond a certain cutoff. (Do character actors have the same
      kind of genre affinity that lead actors do?)
    * In step (2), subtract off the threshhold
    * In step (3), square the individual actor-genre affinity
      scores before adding them
    * Add another pass for second-order affinities: That is,
      any actor who appears in at least (threshold2) movies
      with an actor who has a non-zero actor-genre affinity
      score might be assigned a small positive actor-genre
      affinity score on that basis alone.

'''

import codecs
```

```
from collections import defaultdict

print "Parsing genres"

# First pass: isolate the genre_* columns from our main
# tmdb.tsv table and load them into memory

movie_genres = {}
with codecs.open('data/tmdb.tsv', 'r', 'utf-8') as f:
    tmdb_cols = f.readline().strip().split('\t')
    genre_cols = [c for c in tmdb_cols if c.startswith('genre_')]
    for line in f:
        tmdb_vals = line[:-1].split('\t')
        tmdb_row = dict([z
                         for z in zip(tmdb_cols, tmdb_vals)
                         if z[0].startswith('genre_')])
        movie_genres[int(tmdb_vals[0].strip())] = tmdb_row

# Second pass: For each actor, compute the number of times
# they appeared in a movie with each genre (by adding up
# the movie_genres[] rows for each movie they are listed
# as appearing in).

# Movies in which an actor is credited multiple times (e.g.
# Peter Sellers in Dr. Strangelove) are assumed to have been
# de-duped in the creation of credits-by-name.txt

# You can think of the combination of the first two passes
# as bring roughly
#
# SELECT credit.actor,
#        SUM(movie.genre_*) as actor_genre_*_affinity
#   FROM Credits_By_Name credit
#   JOIN Movie movie
#     ON credit.movie_id = movie_id
#  GROUP BY credit.actor

cred_counts = {}
for g in genre_cols:
    cred_counts[g] = defaultdict(int)

print "Parsing credits"

with codecs.open('data/credits-by-name.txt', 'r', 'utf-8') as f:
    cred_cols = f.readline().strip().split('\t')
    for line in f:
        cred_vals = line[:-1].split('\t')
        cred_name = cred_vals[1]
        for movie_id in cred_vals[3].split(','):
            m = movie_genres[int(movie_id)]
            for g in genre_cols:
                cred_counts[g][cred_name] += int(m[g])
```

```
print "Parsing credits (round 2)"

# Setup for third pass: Create the 2d array in which
# movie_scores[movie_id][genre_name] = 0
# to hold the cast-genre affinity scores for every
# cartesian combination of movie_id and genre_name

movie_scores = {}
for movie_id in movie_genres.keys():
    movie_scores[movie_id] = {}
    for g in genre_cols:
        movie_scores[movie_id][g] = 0

# Define the minimum number of movies of a particular
# genre in which an actor must have appeared for us
# to consider their actor-genre affinity to be worth
# tabulating

threshold = 10

# Third pass: For each movie, accumulate the individual
# actor-genre affinity scores for every actor credited
# as appearing in that movie (provided that the
# actor-genre affinity score exceeds the threshold)

with codecs.open('data/credits-by-name.txt', 'r', 'utf-8') as f:
    cred_cols = f.readline().strip().split('\t')
    for line in f:
        cred_vals = line[:-1].split('\t')
        cred_name = cred_vals[1]
        for movie_id in cred_vals[3].split(','):
            for g in genre_cols:
                if cred_counts[g][cred_name] > threshold:
                    movie_scores[int(movie_id)][g] += cred_counts[g][cred_name]

# Write the output.

# Our primary output file is a tsv that can be joined with
# the main tmdb.tsv file. It contains one row per movie,
# and then (in addition to the tmdb_id join key) one
# column per genre, with the movie's cast-genre affiliation
# score as the value.

# NB: since this Python script depends on the tmdb.tsv file,
# that joining *must* be done in the processing environment.
# For example, in R:
#
# tmdb <- read.table('data/tmdb.tsv',
#                    header=T, sep='\t',
#                    encoding='utf-8',
#                    quote=NULL, comment='')
# cast_affinity <- read.table('data/movie-cast-scores.tsv',
#                             header=T, sep='\t')
```

```
# tmdb_merged <- merge(tmdb, cast_affinity, by='tmdb_id')

print "Writing affinity scores"
with codecs.open('data/movie-cast-scores.tsv', 'w', 'utf-8') as o:
    o.write('tmdb_id\t' + '\t'.join(['score_' + g
                                     for g
                                     in genre_cols]) + '\n')
    for movie_id, movie_score in movie_scores.items():
        o.write(str(movie_id))
        for g in genre_cols:
            o.write('\t%d' % movie_score[g])
        o.write('\n')


# A secondary output file.
#
# So that we can visually inspect the results of this
# data wrangling step, both to manually verify that the
# output looks reasonable and so that we can be inspired
# by looking at it to pursue other approaches, we create
# a tsv of the top performers in each genre. In particular,
# we expect that by better understanding this aspect of
# the processed data we can figure out how to approach
# the question of secondary affinity.

print "Writing list of top performers in each genre"
with codecs.open('data/credit-tiers.tsv', 'w', 'utf-8') as o:
    o.write('Genre\tCount\tName\n')
    for g in genre_cols:
        print g
        sorted_ones = sorted([(v, k)
                              for (k, v)
                              in cred_counts[g].items()
                              if v>=threshold],
                             reverse=True)[:100]
        for (v, k) in sorted_ones:
            o.write('%s\t%d\t%s\n' % (g, v, k))


# Repeat third pass for test data (using the cast affinities
# based only on training data)

# Third pass: For each movie, accumulate the individual
# actor-genre affinity scores for every actor credited
# as appearing in that movie (provided that the
# actor-genre affinity score exceeds the threshold)

print "Computing test data"

movie_scores = {}
with codecs.open('data/credits-by-name-test.txt', 'r', 'utf-8') as f:
    cred_cols = f.readline().strip().split('\t')
    for line in f:
```

```
        cred_vals = line[:-1].split('\t')
        cred_name = cred_vals[1]
        for movie_id in cred_vals[3].split(','):
            if int(movie_id) not in movie_scores.keys():
                movie_scores[int(movie_id)] = {}
                for g in genre_cols:
                    movie_scores[int(movie_id)][g] = 0
            for g in genre_cols:
                if cred_counts[g][cred_name] > threshold:
                    movie_scores[int(movie_id)][g] += cred_counts[g][cred_name]

# Write the output.

print "Writing affinity scores"
with codecs.open('data/movie-cast-scores-test.tsv', 'w', 'utf-8') as o:
    o.write('tmdb_id\t' + '\t'.join(['score_' + g
                                     for g
                                     in genre_cols]) + '\n')
    for movie_id, movie_score in movie_scores.items():
        o.write(str(movie_id))
        for g in genre_cols:
            o.write('\t%d' % movie_score[g])
        o.write('\n')
```

We use the same approach for director affinities.

Some of our graphs above were produced using the following R code:

```
library('cluster')
library('factoextra')
library('mclust')
library('corrplot')
library('dbscan')
library('MASS')
library('ggplot2')
library('ggfortify')
library('NbClust')
library('e1071')
library('tidyr')
library('randomForest') # NB: masks ggplot2::margin
```

```
tmdb <- read.table('data/tmdb.tsv',
                   header=T, sep='\t',
                   encoding = 'utf-8',
                   quote=NULL, comment='')
tmdb$runtime <- as.integer(as.character(tmdb$runtime))
cast_affinity <- read.table('data/movie-cast-scores.tsv',
                            header=T, sep='\t',
                            encoding='utf-8',
                            quote=NULL, comment='')
tmdb <- merge(tmdb, cast_affinity, by='tmdb_id')
```

Split up release date into 3 factors. . . .

```
tmdb$release_year <- as.integer(substr(tmdb$release_date, 1, 4))
tmdb$release_year_4 <- (tmdb$release_year-1) %% 4 + 1
```

```
tmdb$release_year_term <- as.factor(tmdb$release_year -
                                      tmdb$release_year_4 + 1)
tmdb$release_year_4 <- as.factor(tmdb$release_year_4)
tmdb$release_month <- as.factor(substr(tmdb$release_date, 6, 7))
```

Load the test set, too

```
tmdb.test <- read.table('data/tmdb-test.tsv',
                        header=T, sep='\t',
                        encoding = 'utf-8',
                        quote=NULL, comment='')
tmdb.test$runtime <- as.integer(as.character(tmdb.test$runtime))
cast_affinity.test <- read.table('data/movie-cast-scores-test.tsv',
                                 header=T, sep='\t',
                                 encoding='utf-8',
                                 quote=NULL, comment='')
tmdb.test <- merge(tmdb.test, cast_affinity.test, by='tmdb_id')
tmdb.test$release_year <- as.integer(substr(tmdb.test$release_date,
                                            1, 4))
tmdb.test$release_year_4 <- (tmdb.test$release_year-1) %% 4 + 1
tmdb.test$release_year_term <- as.factor(tmdb.test$release_year - tmdb.test$release_year_4 + 1)
tmdb.test$release_year_4 <- as.factor(tmdb.test$release_year_4)
tmdb.test$release_month <- as.factor(substr(tmdb.test$release_date,
                                            6, 7))
tmdb.test$revenue <- as.integer(tmdb.test$revenue)
```

```
tmdb_budget = tmdb[tmdb$budget > 0,]
median_budget <- aggregate(as.integer(tmdb_budget$budget),
                           by=list(tmdb_budget$release_year_term),
                           median)
```

```
box.plot.budget <- function(genre) {
  print(genre)
  ggplot(tmdb_budget[tmdb_budget[,paste0('genre_', genre)] == 1,],
         aes(x=release_year_term, y=budget)) +
    geom_boxplot() +
    geom_line(data=median_budget, aes(x=as.numeric(Group.1), y=x),
              color='red') +
    scale_x_discrete(breaks=seq(1917, 2017, 4), drop=FALSE) +
    scale_y_log10(limits=c(1, 1e+9),
                  breaks=c(1e2, 1e4, 1e6, 1e8),
                  labels=c('$100', '$1K', '$1M', '$100M')) +
    labs(title=genre, x='Release Year', y='Budget') +
    theme(axis.text.x = element_text(angle = 90,
                                     hjust = 1, vjust=0.4))
}
box.plot.budget('Action')
```

```
lapply(lapply(c("Action", "Adventure", "Animation",
                "Comedy", "Crime", "Documentary", "Drama",
                "Family", "Fantasy", "Foreign",
                "History", "Horror",
                "Music", "Mystery", "Romance", "Science_Fiction",
                "TV_Movie", "Thriller", "War", "Western"),
              box.plot.budget), invisible)
```

```
box.plot.score <- function(genre) {
  print(genre)
  tmdb.test$genre_score <- tmdb.test[,paste0('score_genre_', genre)]

  ggplot(tmdb.test,
         aes(x=release_year_term,
             y=genre_score,
             fill=(tmdb.test[,paste0('genre_', genre)] == 1))) +
    geom_boxplot() +
    labs(title=genre, x='Release Year', y='Genre Score') +
    guides(fill=F) +
    theme(axis.text.x = element_text(angle = 90,
                                     hjust = 1, vjust=0.4))
}
box.plot.score('Romance')
```

How about director affinity?

```
director.affinity.test <- read.table(
  'data/movie-director-scores-test.tsv',
  header=T, sep='\t')
box.plot.dscore <- function(genre) {
  print(genre)
  # quick one-off JOIN between two data frames
  tmdb.test$genre_score <-
    director.affinity.test[match(tmdb.test$tmdb_id,
                                 director.affinity.test$tmdb_id),
                           paste0('score_genre_', genre)]

  ggplot(tmdb.test,
         aes(x=release_year,
             y=genre_score,
             color=(tmdb.test[,paste0('genre_', genre)] == 1))) +
    geom_point(alpha=0.8) +
    labs(title=genre, x='Release Year', y='Genre Score') +
    guides(color=F) +
    theme(axis.text.x = element_text(angle = 90, hjust = 1, vjust=0.4))
}
box.plot.dscore('Romance')
box.plot.dscore('Thriller')
```