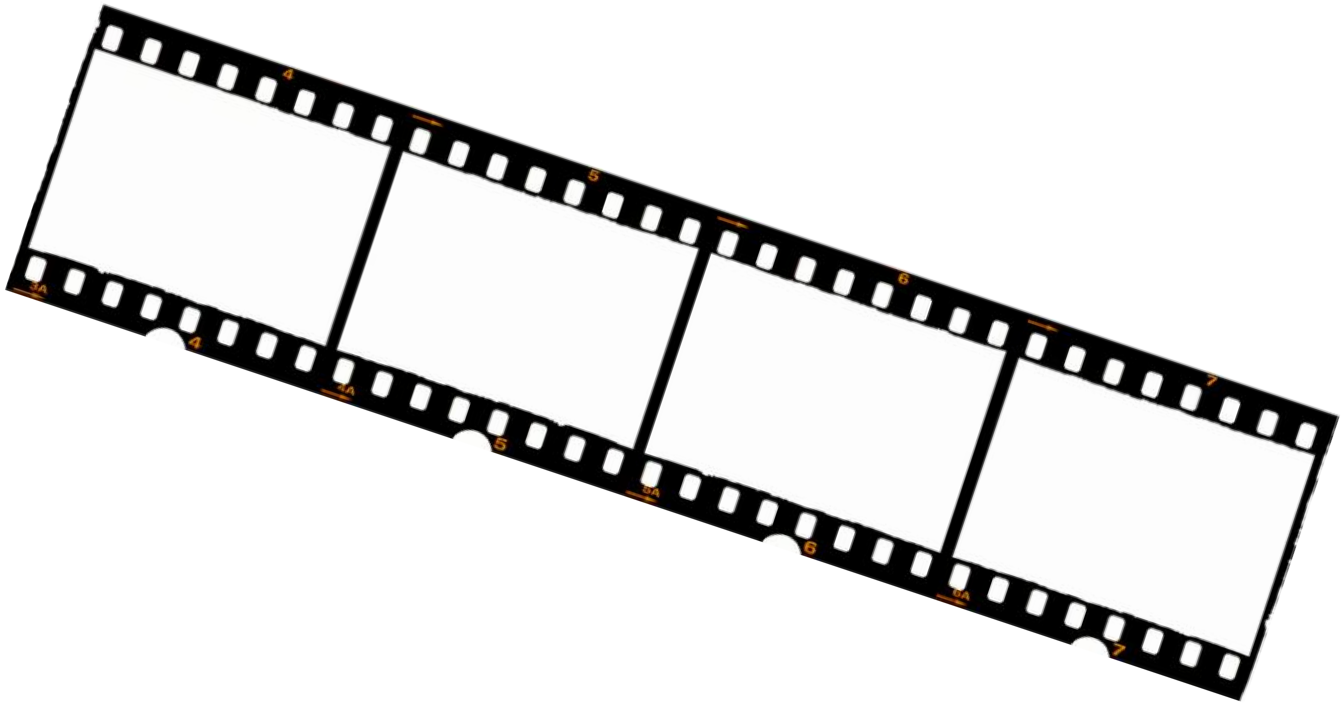
Prediction of a films success

Using a dataset of ~45,000 movies



Name: Celine Stenberg
Supervisor: Michael Claudius
School: Zealand - Sjællands Erhversakademi

Deadline: 29-05-2020
Count: words - 4133 (without front page, table of contents, and bibliography)
characters - 23561 (without front page, table of contents, and bibliography)

1. Introduction and motivation **3**
2. Problem definition **3**
3. Method **3**
4. Planning **4**
5. Process / Findings **5**
5.1. Inspection 5
5.2. Cleaning 6
5.2.1. Excel and VSCode 7
5.2.2. Creating the new datasets 8
5.2.3. Updating the movies_metadata dataset with new attributes 9
5.3. Small movies dataset with revenue as label 11
5.3.1. Preparing movies_small 11
5.3.2. Models 14
Linear Regression 14
Batch Gradient Descent 14
Results on the validation set 15
Random Forest 15
Fine-tuning 15
Results on the validation set 16
Functional API 16
Fine tuning 17
Results on the validation set 18
5.4. Final predictions for all models 18
5.4.1. Linear Regression 19
5.4.2. Batch Gradient Descent 19
5.4.3. Random Forest 19
5.4.4. Functional API 20
6. Conclusion **21**
6.1. What should the model predict? 21
6.2. What model should be used to predict success? 21
6.3. What attributes seem to be most relevant? 21
6.4. Can the model be used by filmmakers? 22
7. Reflection **22**
7.1. The questions 22
7.2. Method 22
7.3. Plan 22
8. Bibliography **23**

1. Introduction and motivation

Filmmaking is an enormous and far reaching global industry. Cinemas, award shows and every day tv has been and are still a big part of our lives. We consume film like a sustenance for our well-being and they form us and our memories with their portrayals and influence on pop-culture. So, being such a big part of our lives today, one would think that there must be a recipe for predicting, and thereby securing, a release's success. It's an interesting thought, and one that can be applied to many scenarios in our day-to-day lives.

In this synopsis I will attempt to make a model that does just that.

2. Problem definition

The big question is then:

Is it possible to correctly predict an unpublished films success based on data collected from already released films?

This further leads to the sub-questions:

- What defines a films success - what should the model predict?
- What model should be used to predict success?
- What attributes / combination of attributes seem to be most relevant?
- And can this be used by filmmakers to ensure higher success?

3. Method

When answering these questions, I intend to mainly rely on practical and experimental work. This is simply because I'm handling a specific and individual case and not a common problem or topic often discussed. For this reason I will mostly have a somewhat trial-and-error approach when figuring out what to proceed with in this project.
However, the different models and methods that can be used in similar cases are of course explained online and in books. So, I will naturally delve into researching online as well as reading up on topics in our textbook[1] and re-watching videos for this curriculum when dealing with questions not specifically pertaining to this individual case. By doing so I can compare methods and their usefulness in general - and from there decide what to try out.

[1] This semester has largely followed the book *Hands-on Machine Learning with Scikit-Learn, Keras and Tensorflow* by Aurélien Géron

4. Planning

I will in general write about and explain interesting findings and important points of the process in the synopsis throughout the week. Should I forget, then the plan for the weekend is specified as a reminder. This way the synopsis will be up to date at the beginning of every week.

I will start by examining the dataset then move on to preparing it for training. I will also have time set aside for studying and testing different approaches.

<table>
<tr><th></th><th>Mon.</th><th>Tue.</th><th>Wed.</th><th>Thur.</th><th>Fri.</th><th>Sat.</th><th>Sun.</th></tr>
<tr><td>Week 18</td><td colspan="5">● Examining the dataset and defining the project’s goal and framework / structure.
● Manipulating and cleaning the dataset making it ready for further inspection.
● Setting up / writing the theoretical parts of the synopsis.</td><td colspan="2" rowspan="4">Same as the weekdays + making sure that the synopsis is updated.</td></tr>
<tr><td>Week 19</td><td colspan="5">● Visualizing the data.
● Looking for correlations.
● Further cleaning of the dataset.
● Testing attribute combinations.</td></tr>
<tr><td>Week 20</td><td colspan="5">Researching and experimenting with:
● Training and evaluating.
● Fine tuning.</td></tr>
<tr><td>Week 21</td><td colspan="5">● Training and evaluating.
● Fine tuning.</td></tr>
<tr><td>Week 22</td><td colspan="4">● Finishing the synopsis.
● Preparing for the oral exam.</td><td>Deadline Fri. 22/5 11:00</td><td colspan="2"></td></tr>
</table>

5. Process / Findings

5.1. Inspection

The data I will be using is found on kaggle.com as *the Movies dataset*. The set is made up of 7 .csv files containing different information relating to movies. I will be using two of the files: *movies_metadata.csv* that handles specific information such as name, budget, revenue,

average voting scores, and genres as well as *credits.csv* that contains all the information for the cast and the crew of each film.

Movies_metadata has 24 columns but many of them seem quite irrelevant. These include for example the columns homepage, poster_path, and video. Several of the columns also have clear missing values portrayed with either 0's or empty lists/strings. Columns where there can be multiple values attached to a specific film are JSON arrays inside strings.

	adult	belongs_to_collection	budget	genres	homepage	id	imdb_id
0	False	{'id': 10194, 'name': 'Toy Story Collection', ...	30000000	[{'id': 16, 'name': 'Animation'}, {'id': 35, '...	http://toystory.disney.com/toy-story	862	tt0114709
1	False	NaN	65000000	[{'id': 12, 'name': 'Adventure'}, {'id': 14, '...	NaN	8844	tt0113497
2	False	{'id': 119050, 'name': 'Grumpy Old Men Collect...	0	[{'id': 10749, 'name': 'Romance'}, {'id': 35, ...	NaN	15602	tt0113228

Running the .corr() function shows numerical values correlation to each other. So far revenue and vote_count seem to be the only attributes correlating. It also shows that some of the numerical datas types are wrong as they're not shown in the table.

```
movies_metadata.corr()
```

	revenue	runtime	vote_average	vote_count
revenue	1.000000	0.103917	0.083868	0.812022
runtime	0.103917	1.000000	0.158146	0.113539
vote_average	0.083868	0.158146	1.000000	0.123607
vote_count	0.812022	0.113539	0.123607	1.000000

Using the function type() shows that for example the data in the column *budget* are of type string, and not int as it should be. Trying to cast budget as a type int returns an error message saying that "/ff9qCepilowshEtG2GYWwzt2bs4.jpg" could not be converted. This shows that it was not just put in as the wrong type, but something else has happened.

It's thus clear that the dataset will need a bit of cleaning up before proceeding further. However, just looking at the information available, it seems reasonably logical to use revenue as the indicator of success. Revenue represents the value of a film in relation to its income and hints to the amount of people that has paid to watch it.
Other attributes that may be interesting to explore are the genres as well as the vote_count and vote_average. The last two, as well as revenue, could be used to make a form of scoring system for each genre. Likewise, with the credits.csv file containing information about the cast, it could be interesting to see if the actors in a movie correlates to the revenue. This could be done similarly with a scoring system for every actor themselves as well as a cumulative score for the films.

Now, having a label that the model will be trained on and no continuous flow of data indicates that supervised offline learning will be the best approach when creating a model to complete the task.

5.2. Cleaning

As mentioned there is a problem with the types of data in some of the columns. After looking at the data in VSCode it was clear that the error message was due to a new line that had been made inside a string. This had been interpreted as dividing two instances and therefore putting a string value belonging to another column into *budget*. In an attempt to find more of these mistakes I made a function that would check if the value was an int, and if it was not, return the value as well as the index.

Next I tackled the problem with the JSON arrays. Because of the way the values are set up it's not possible to easily access all the information and also compare them to other values in other columns. Each movie is in it's own array inside a JSON string and has many mistakes so that even built in functions such as json.load(), that reads data in JSON formats, are not usable.
A way to solve this issue is creating new .csv files with all the data inside one pair of brackets and in the correct JSON format.
I will not be doing this with all JSON array columns, but only genres from movies_metadata.csv and cast from credits.csv as they are the ones of interest.

5.2.1. Excel and VSCode

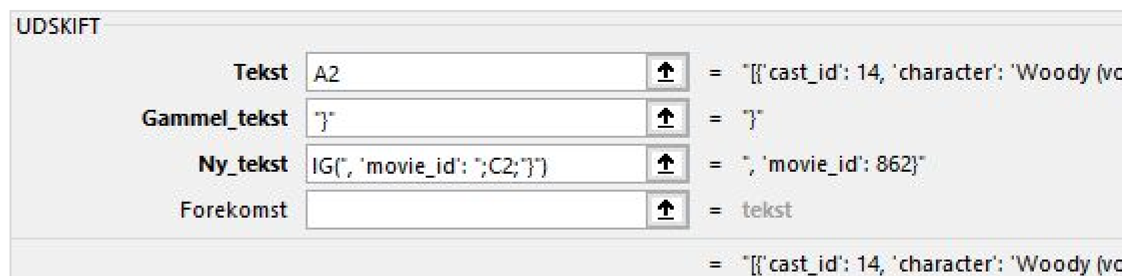
When making the new cast and genres_instances datasets, the movie_id the data is connected to still needs to be included for each mention of a genre or actor. There will therefore have to be a column with "movie_id" so the values can be accessed based on the movie. A manageable way to do this is to make the new dataset in Excel. Excel displays the actual data, has a number of functions available and eliminates some of the problems that could occur using python.

First I imported the data as a .csv file with UTF-8 encoding. This ensures that each column in the array actually gets its own column in Excel and with the correct characters. The dataset has

3 columns: *cast*, *crew*, and *movie_id*. I then used the SUBSTITUTE (UDSKIFT) function in a new column so I could take the values in the cast column (column A) and replace the closing curly brackets with the movie_id value (column C) + the closing curly brackets. This I did with the CONCATENATE (SAMMENKÆDNING) function. It basically looks like this:

```
=UDSKIFT(A2;"}";SAMMENKÆDNING(", 'movie_id': ";C2;"}"))
```

Or like this if using the built-in function window:



Now the new column has all the information in the cast column with the added movie_id from the movie_id column. The new column is the only one needed for the new dataset so the original columns need to be deleted. But the newly made column is dependent on variables referencing them so first the new column has to be copied and pasted into another column as their values without references to other columns. Each row is still one movie made up of a JSON array containing multiple JSON strings.

The file is then saved as a .txt file and opened in VSCode where I'm separating each reference to a cast member by removing the brackets around each row, besides the first bracket pair in the very beginning and the very end of all the data, and changing the single quotes to JSON appropriate double quotes. This is all done with the basic find and replace function.
I then tried to load the file with json.load() in jupyter. Whenever an error came up I would find it in VSCode and search for that general type of error to correct all instances of it.

5.2.2. Creating the new datasets

With a clean file it is now possible to open it in jupyter as a json file. After changing it to a pd.DataFrame I casted it to it's own .csv in the proper directory. This is done with both cast and genres like so:

```
with open("Movie_Dataset/cast_text_v3.txt", "rb") as f:
    data = json.load(f)

cast = pd.DataFrame(data)
cast.to_csv("Cleaned_Dataset/cast.csv", index_label=False)
cast = pd.read_csv("Cleaned_Dataset/cast.csv")
```

Next the actual datasets for actors and genres need to be made. To do this I merged each actor with vote_average from *cast* and revenue from *movies_metadata* on the id of the film. I grouped the actors by their id, so that the same actor was in it's own group of all the films they had starred in, and then used the .agg function to put in several values, such as the mean and the sum, based on the vote_average and revenue of the films. This is how:

```
movies = pd.read_csv("movies_metadata_pipe_v1.csv", delimiter="|")
cast_with_scores = cast.merge(movies[["id", "vote_average", "revenue"]],
                              left_on="movie_id", right_on="id", left_index=True)

actors = cast_with_scores[["id_x", "name", "vote_average", "revenue"]]
actors = actors.rename(columns={"id_x": "id"})

actors = actors.merge(actors.groupby("id")["vote_average"]
    .agg(["sum","mean", "median", "count", "min", "max"]), left_on="id", right_index=True)
del actors['vote_average']
actors = actors.rename(columns={"sum": "vote_sum", "mean": "vote_mean", "median": "vote_median",
                               "count": "vote_count", "min": "vote_min", "max": "vote_max",})

actors = actors.merge(actors.groupby("id")["revenue"]
    .agg(["sum","mean", "median", "count", "min", "max"]), left_on="id", right_index=True)
del actors['revenue']
actors = actors.rename(columns={"sum": "revenue_sum", "mean": "revenue_mean",
                                "median": "revenue_median", "count": "revenue_count",
                                "min": "revenue_min", "max": "revenue_max",})

actors = actors.drop_duplicates(subset = ["id"])
```

So now a dataset with each respective actor and their scores can be used in the movies_metadata set. Of course this is also done with genres and both are saved as .csv files, ready to use for new attributes. A slice of what's in the dataset can be seen here:

id	name	vote_sum	vote_mean	vote_median	vote_count	vote_min	vote_max	revenue_sum	revenue_mean
31	Tom Hanks	477.0	6.625000	6.8	72	4.0	8.4	9936142170	1.380020e+08
12898	Tim Allen	178.4	6.151724	6.3	29	4.5	7.7	3305184195	1.139719e+08
7167	Don Rickles	187.9	6.479310	6.8	29	0.0	7.8	2097375480	7.232329e+07

5.2.3. Updating the movies_metadata dataset with new attributes

Before rechecking the correlation between the numerical values, the new attributes should be added. The attributes I want to try out are:

- cast_vote_sum
- cast_vote_mean
- cast_revenue_sum
- cast_revenue_mean
- genres_vote_sum
- genres_vote_mean
- genres_revenue_sum
- genres_revenue_mean

Just like when making the new datasets, the new attributes are made by first merging the individual actors' scores with the cast dataset and then merging, grouping and aggregating the chosen values into the movies_metadata dataset. The exact same is done with genres.

```
cast = cast.merge(actors[["id", "vote_mean", "revenue_mean"]], on="id", left_index=True)

movies_cast_score = movies.merge(cast.groupby("movie_id")["vote_mean"]
    .agg(["sum", "mean"]), left_on="id", right_index=True)
movies = movies_cast_score.rename(columns={"sum": "cast_vote_sum", "mean": "cast_vote_mean"})

movies_cast_score = movies.merge(cast.groupby("movie_id")["revenue_mean"]
    .agg(["sum", "mean"]), left_on="id", right_index=True)
movies = movies_cast_score.rename(columns={"sum": "cast_revenue_sum", "mean": "cast_revenue_mean"})
```

With the new attributes and the data types corrected, the .corr() function can be re-used.

	adult	budget	id	popularity	revenue	runtime	video	vote_average	vote_count
adult	1.000000	-0.003244	0.005070	-0.002932	-0.002379	-0.002607	-0.000521	-0.015655	-0.002837
budget	-0.003244	1.000000	-0.094965	0.443838	0.764501	0.135257	-0.010028	0.070510	0.669585
id	0.005070	-0.094965	1.000000	-0.062630	-0.066287	-0.097573	0.033980	-0.132551	-0.056655
popularity	-0.002932	0.443838	-0.062630	1.000000	0.503815	0.117561	-0.013238	0.149442	0.557653
revenue	-0.002379	0.764501	-0.066287	0.503815	1.000000	0.104499	-0.007309	0.086393	0.808196
runtime	-0.002607	0.135257	-0.097573	0.117561	0.104499	1.000000	-0.003712	0.147038	0.113017
video	-0.000521	-0.010028	0.033980	-0.013238	-0.007309	-0.003712	1.000000	-0.011412	-0.008608
vote_average	-0.015655	0.070510	-0.132551	0.149442	0.086393	0.147038	-0.011412	1.000000	0.129511
vote_count	-0.002837	0.669585	-0.056655	0.557653	0.808196	0.113017	-0.008608	0.129511	1.000000
cast_vote_sum	-0.010343	0.353287	-0.105065	0.364961	0.358100	0.197249	-0.021740	0.214571	0.420649
cast_vote_mean	-0.023649	0.061359	-0.099976	0.118893	0.069562	0.131404	-0.021546	0.813811	0.102985
cast_revenue_sum	-0.002164	0.526158	-0.015112	0.482330	0.733876	0.083373	-0.005186	0.060679	0.638149
cast_revenue_mean	-0.005729	0.647885	-0.033043	0.516831	0.787104	0.077072	-0.007313	0.071018	0.685241
genres_vote_sum	-0.001889	0.180913	-0.169873	0.161848	0.132082	0.102402	-0.004456	0.054057	0.129901
genres_vote_mean	-0.004521	-0.023767	0.023946	-0.041988	-0.018710	0.070767	0.011891	0.176033	-0.034537
genres_revenue_sum	-0.004927	0.292783	-0.138576	0.212433	0.236271	0.019886	0.006053	0.007322	0.214431
genres_revenue_mean	-0.004642	0.247169	-0.091342	0.168397	0.202970	-0.040939	0.006452	-0.036951	0.182097

As expected, *budget* and *revenue* does correlate well at 0.764501. As can be expected revenue also correlates great with the new attribute cast_revenue_mean at 0.787104.
As mentioned the vote_count also correlates really well with revenue, but since it's a value that will only be known after the release of a movie we can't really use it in the model. So, all attributes that cannot be known before a release will be removed with the exception of the label. However, the attribute vote_average correlates even more with cast_vote_mean than revenue does with budget or cast_revenue_mean. Using *vote_average* as an alternative success indicator / label with *cast_vote_mean* as the main predictor seems to be very much a possibility too.
The attributes with no interest will be removed before progressing further. The attributes that have been kept are these:

- budget
- revenue
- vote_average
- vote_count
- cast_score_sum
- cast_score_mean
- cast_revenue_sum
- cast_revenue_mean
- genres_score_sum
- genres_score_mean
- genres_revenue_sum
- genres_revenue_mean

Before the dataset can be used it needs to be scaled. This includes the label revenue, as it's returning numbers in the millions and therefore is of a completely different range that might skew the results. I am using MinMaxScaler() for this on the whole dataset like so:

```
scaler = MinMaxScaler()
scaler.fit(movies_removed_att)
movies_scaled = scaler.transform(movies_removed_att)
movies_scaled = pd.DataFrame(movies_scaled, index=movies_removed_att.index,
                             columns=movies_removed_att.columns)
```

5.3. Small movies dataset with revenue as label

5.3.1. Preparing movies_small

All the following models are going to use *revenue* as the label. The relevant attributes for this task have been copied from the movies_scaled dataset into another named movies_small. The dataset has quite a lot of missing values, represented with "0", and to not have the calculations be too skewed, each row with a 0 in *revenue* or *budget* is removed. This removes a substantial amount of the dataset and leaves only 5347 movies. This is how the data looks now:

budget	revenue	cast_vote_sum	genres_vote_sum	cast_revenue_sum	genres_revenue_sum
0.078947	1.339881e-01	0.079305	0.236371	1.209510e-02	0.207527
0.171053	9.426131e-02	0.160671	0.226483	2.373239e-02	0.338819
0.042105	2.921563e-02	0.056900	0.230300	2.962147e-03	0.075689

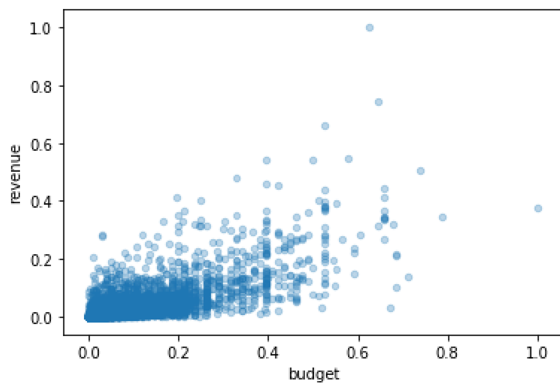
The dataset is then randomized and split into a training set, validation set(20% of the training set), and a test set (20% of the whole set). *Revenue* is removed and put into its own corresponding datasets.

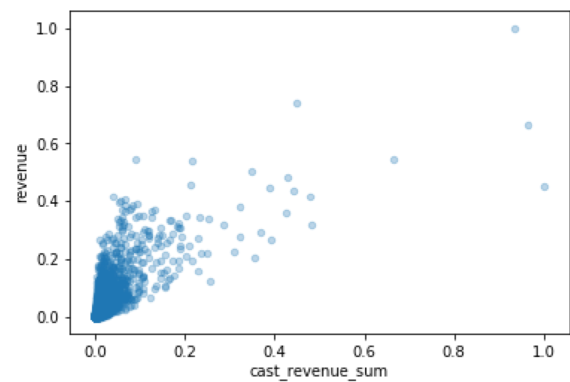
```
small_split_train_set, small_split_test_set = train_test_split(movies_small_scaled,
                                                               test_size=0.2, random_state=42)

X_train_full = small_split_train_set.drop("revenue", axis=1)
Y_train_full = small_split_train_set["revenue"].copy()

X_valid, X_train = X_train_full[:860], X_train_full[860:]
Y_valid, Y_train = Y_train_full[:860], Y_train_full[860:]

print(len(X_train), len(X_valid), len(small_split_test_set))

#variables for the test set
X_test = small_split_test_set.copy()
X_test = X_test.drop("revenue", axis=1)
Y_test = small_split_test_set["revenue"].copy()
```

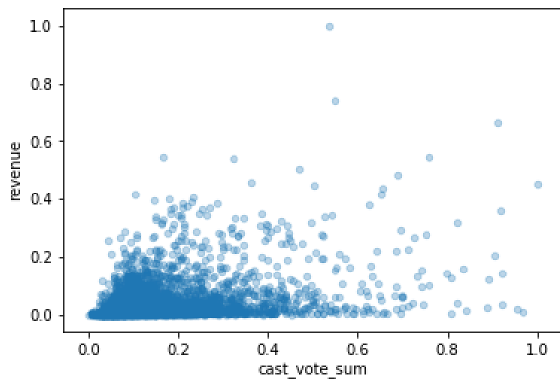
Of course removing the instances of 0 changes the correlation between some.

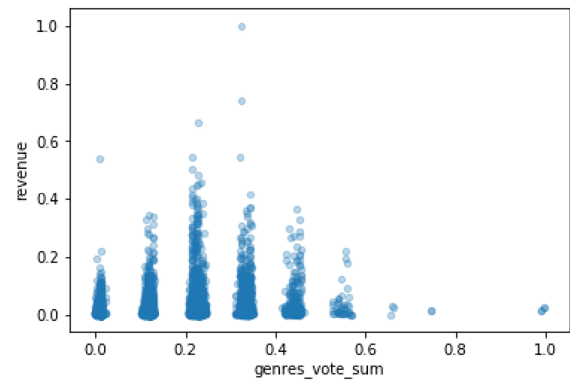
	budget	revenue	cast_score_sum	genres_score_sum	cast_revenue_sum
budget	1.000000	0.725706	0.299620	0.226836	0.518802
revenue	0.725706	1.000000	0.373085	0.155739	0.750349
cast_score_sum	0.299620	0.373085	1.000000	0.053019	0.614440
genres_score_sum	0.226836	0.155739	0.053019	1.000000	0.094935
cast_revenue_sum	0.518802	0.750349	0.614440	0.094935	1.000000
genres_revenue_sum	0.443413	0.351559	0.079451	0.766241	0.216929

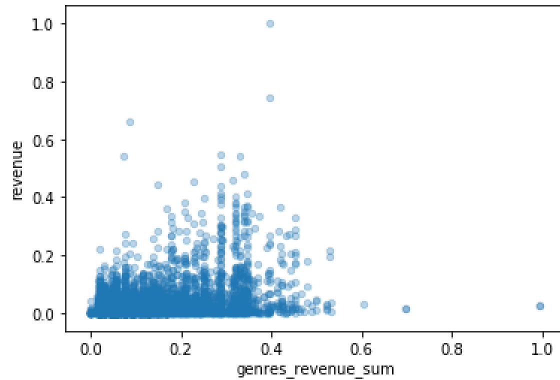
Plotting the attributes against each other will show the correlation more visually. None of them seem to have a clear linear tendency, but as expected *budget* and *cast_revenue_sum* are the closest to one.











In all of the models that I will test I will use MAE (Mean Absolute Error) as the loss function due to the amount of outliers. Understanding what the error means can be a bit difficult with big or scaled numbers. Using the .describe() function helps with putting the predictions and the MAE values in a better perspective. They can now be compared to f.x the mean of *revenue*.

	budget	revenue	cast_vote_sum	genres_vote_sum	cast_revenue_sum	genres_revenue_sum
count	5.347000e+03	5.347000e+03	5347.000000	5347.000000	5.347000e+03	5347.000000
mean	8.155705e-02	3.223770e-02	0.137873	0.184688	1.299800e-02	0.126940
std	1.048032e-01	5.921426e-02	0.115470	0.122885	3.869116e-02	0.104101
min	2.631579e-09	3.586845e-10	0.000963	0.000000	9.642181e-08	0.000755
25%	1.382895e-02	2.539089e-03	0.071076	0.116470	2.052065e-03	0.049424
50%	4.473684e-02	1.074260e-02	0.101036	0.214980	5.066254e-03	0.092458
75%	1.052632e-01	3.564978e-02	0.159967	0.232313	1.030573e-02	0.177055
max	1.000000e+00	1.000000e+00	1.000000	0.995791	1.000000e+00	0.992646

For this reason I made a function that calculates the percentage error of the predictions' absolute mean value against the absolute mean value of *revenue*. This is not a true percentage error for every instance, like MAPE (Mean Absolute Percentage Error), but instead the average MAE compared to the average revenue.

```
def mean_percentage_error(y_true, y_pred):
    mae = mean_absolute_error(y_pred, y_true)
    label_mean = np.mean(np.abs(y_true))
    mpe = mae / label_mean
    return mpe
```

5.3.2. Models

Linear Regression

The first model to try out is the simple Linear Regression model. The model is fitted to the training set and is then used to predict on the validation set.

```
MAE on validation set: 0.01672690592255436
MAE in percentage:  0.574148109402754
R2 score for the features: 0.5969796915553094
```

Batch Gradient Descent

Using the optimization tool gradient descent is a way of training a Linear Regression model. The gradient descent will actually create its own model and so it can be interpreted as a separate model entirely. In this case I will be using Batch Gradient Descent (BGD). With BGD it's possible to add more tunable parameters, such as biases and learning rate, to the model, and it will test them on the whole training set in iterations. For each iteration the parameters are changed to improve on the loss function. The BGD function will return the history of the iteration results and the finished model with tuned parameters.
Here's the batch_gradient_descent[2] function:

```
def batch_gradient_descent(X, Y, learning_rate, n_iterations):
    weights = np.zeros(X.shape[1])
    history = []
    m = len(Y) #number of training examples

    predict = lambda x: np.dot(x, weights)
    derivative = lambda loss: (X.T.dot(loss)) / m

    for i in range(n_iterations):
        predictions = predict(X)
        loss = predictions - Y
        weights = weights - learning_rate * derivative(loss)

        if i % 50 == 0:
            history.append(mean_absolute_error(X.dot(weights), Y))
    return predict, history
```

To test as many parameters as possible I've made a bgs_test[3] function that takes an array of iterations and learning rates and runs them against each other in the BGD function.

[2] http://pavelbazin.com/post/linear-regression-hyperparameters/
[3] http://pavelbazin.com/post/linear-regression-hyperparameters/

```
def bgd_test(X, Y):
    record = []

    for i in n_iterations:
        for j in learning_rates:
            weights, records = batch_gradient_descent(X, Y, j, i)
            record.append(dict(learning_rates=j, n_iterations=i,
                               w=weights, history=records))
    return record
```

The arrays that I will be testing are these:

```
learning_rates = [.5, .1, .01, .001, .0001]
n_iterations = [30000, 40000, 50000]
```

The test showed that the best parameters were a learning rate of 0.1 and 40000 iterations.

Results on the validation set

With the best parameters in hand a new model can be trained with them on the training set. The fine tuned model is then saved as *predictor* and used on the validation set.

```
MAE on validation set:  0.016217384468791937
MAE in percentage:  0.556658881602059
R2 score for the features: 0.6569252849220324
```

Both the MAE and R2 score seems to be slightly better than with the Linear Regression model.

Random Forest

The Random Forest model is used similarly to the Linear Regression model; it's trained and then used on the validation set. The results are as so:

```
MAE on validation set: 0.012327139596717397
MAE in percentage:  0.4231269026422731
R2 score for the features: 0.8323904420590645
```

Fine-tuning

So far the Random Forest model performs significantly better compared to the previous models. To fine tune the model I've chosen GridSearchCV. To do this a parameter grid with arrays of Random Forest parameters has to be declared. Next the GridSearchCV function[4] is used with the estimator set to a Random Forest model, the parameter grid, the number of cross validation

[4] Aurélien Géron, p. 76

folds and the wanted performance measure. When the grid_search has been made it's fitted on the training set.

```
param_grid = [
    {'n_estimators': [50, 60, 70, 80, 90, 100, 110], 'max_features': [1, 2, 3, 4, 5]},
    {'bootstrap': [False], 'n_estimators': [10, 50], 'max_features': [1, 3, 5]},]

forest_reg = RandomForestRegressor(criterion="mae")
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring="neg_mean_absolute_error", return_train_score=True)

grid_search.fit(X_train_full, Y_train_full)
```

.best_params_ showed that a max features at 5 and number of estimators at 100 is the most optimal parameter settings.

```
print("Best Estimator:", grid_search.best_estimator_)
```

```
Best Parameter: {'max_features': 5, 'n_estimators': 100}
```

Results on the validation set

Using a Random Forest model with the specified parameters yields these numbers on the validation set:

```
MAE on validation set: 0.01195340628072969
MAE in percentage:  0.410298572179445
R2 score for the features: 0.8439231366565587
```

Functional API

The last model I'm going to try is a Functional API.
First the model should be defined. It will just be a simple model with 1 hidden layer with 10 neurons for now. The activation function will be relu as it's usually the default. The models structure[5] looks like this:

```
input_ = keras.layers.Input(shape=(3, ))
hidden1 = keras.layers.Dense(10, activation="relu")(input_)
concat = keras.layers.Concatenate()([input_, hidden1])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])
```

[5] Aurélien Géron, p. 309

Next step is to compile the model. The loss function will be MAE just like with the other models. The optimizer will be Adam, simply because it's faster than stochastic gradient descent (SGD) that I would otherwise have used. SGD functions just like the Batch Gradient Descent from earlier, but will only train on a single instance at a time compared to the whole training set at once. The metrics will be MAE and MAPE (Mean Absolute Percentage Error). The metrics are used when evaluating a model and will show the error scores for each epoch for both the training set and validation set.
The model is fit on the training set, validated on the validation set and finally evaluated on the test set.

```
model.compile(loss="mean_absolute_error", optimizer="adam", metrics=["mae","mape"])

history = model.fit(X_train, Y_train, epochs=50, validation_data=(X_valid, Y_valid))
mae_test = model.evaluate(X_test, Y_test)
```

And the predictions on the validation set looks as follows:

```
MAE on validation set: 0.014093036212382328
MAE in percentage:  0.5147955897564567
R2 score for the features: 0.7293229667223147
```

Fine tuning

The fine tuning of this model is basically making a function that creates a functional API model and compiles it and then an excessive amount of for-loops taking pre-defined arrays as the parameters to try against each other.
The function[6] that will build the model is defined like this:

```
def build_funcAPI_model(n_neurons=[10], learning_rate=0.01, input_shape=5):
    model = keras.Model()
    input_ = keras.layers.Input(shape=(input_shape, ))

    x_hidden = keras.layers.Dense(n_neurons[0], activation="relu")(input_)
    for i in range(1,len(n_neurons)):
        x_hidden = keras.layers.Dense(n_neurons[i], activation="relu")(x_hidden)

    concat = keras.layers.Concatenate()([input_, x_hidden])
    output = keras.layers.Dense(1)(concat)
    model = keras.Model(inputs=[input_], outputs=[output])

    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    model.compile(loss="mae", optimizer=optimizer, metrics=["mae", "mape"])
    return model
```

[6] Aurélien Géron, p. 320

I'm going to try a large number of arrays against each other. It will take a long time, but will only have to be done once. The arrays are specified here:

```
neurons = [[10], [10, 30, 10], [50, 30, 10], [128, 512]]
learning_rates = [1, .1, .01, .001, .0001]
n_epochs = [50, 100, 200, 300, 400, 500]
results = []
```

And lastly the for loops that will do all the work:

```
for i in n_epochs:
    for j in learning_rates:
        for k in neurons:
            model = build_funcAPI_model(k, j, 5)
            history = model.fit(X_train, Y_train, epochs=i,
                                validation_data=(X_valid, Y_valid))
            mae_test = model.evaluate(X_test, Y_test)

            results.append(dict(epochs=i, learning_rates=j, neurons=k,
                                cost=mae_test[1], cost_percent=mae_test[0]))
            print("epochs:", i, "learning rate:", j, "neurons:", k)
            print(" ")
```

Sorting results by the MAE amount will show the best parameters which turned out to be 400 epochs, a learning rate at 0.001 and three hidden layers with 10, 30, 10 neurons.

Results on the validation set

Now the model can be used to predict on the validation set. This is how it turned out:

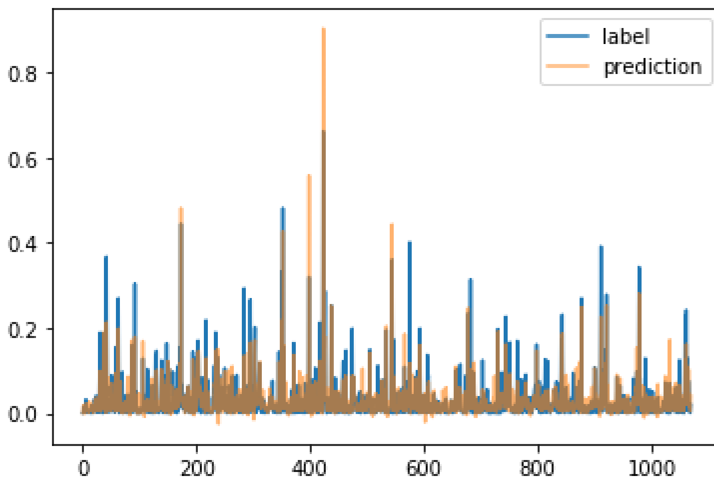
```
MAE on validation set: 0.014093036212382328
MAE in percentage:  0.44149938599839483
R2 score for the features: 0.8013367481945768
```

5.4. Final predictions for all models

Because it's such a small dataset it would be interesting to see how well all of the models do on the test set that's normally reserved for the best. Even though the Batch Gradient Descent model is an optimization of the Linear Regression model I do not consider them as one. Therefore I will include the final prediction scores for the pre-optimized Linear Regression. The histograms under every result shows the predictions on top of the label. The x-axis is the number of instances in the test set while the y-axis is the predicted amount. The histograms show how the models predicted and exactly where they were wrong.

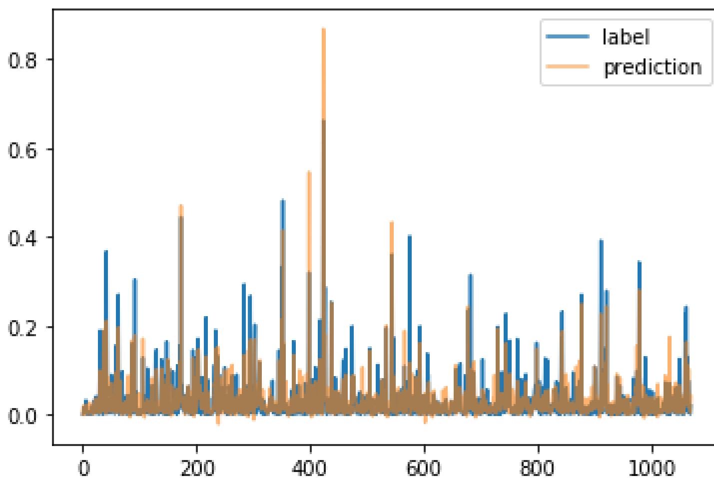
5.4.1. Linear Regression

```
final MAE on test set: 0.017504776225740058
final MAE in percentage:  0.5322887123155582
R2 score for the features: 0.73876203844995
```



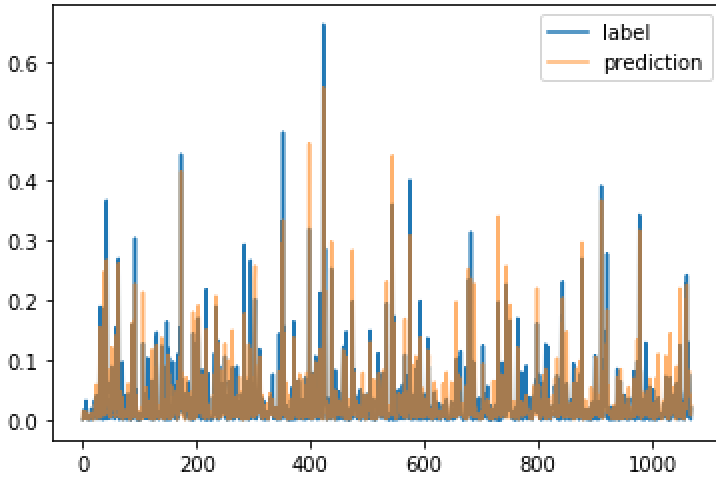
5.4.2. Batch Gradient Descent

```
final MAE on test set: 0.01749809696792419
final MAE in percentage:  0.532085608117242
R2 score for the features: 0.7413409824548668
```



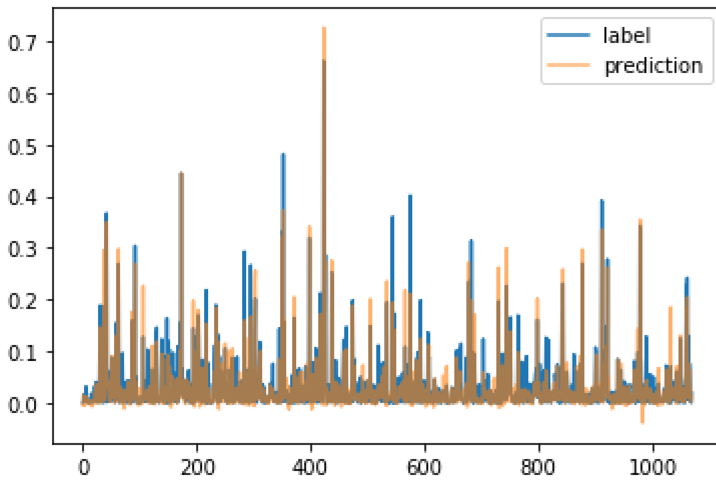
5.4.3. Random Forest

```
final MAE on test set: 0.014117856646988435
final MAE in percentage:  0.42929858904628254
R2 score for the features: 0.8327383373067601
```



5.4.4. Functional API

```
final MAE on test set: 0.013688536269600114
final MAE in percentage:  0.41624372973795126
R2 score for the features: 0.8412600804260673
```



Here are all the results put into a table for easier comparisons:

	Final MAE on test set	**Final MAE in percentage**	**Final R2 on test set**
Linear Regression	0.017504776225740058	0.5322887123155582	0.73876203844995
Batch Gradient Descent	0.01749809696792419	0.532085608117242	0.7413409824548668

Random Forest	0.014117856646988435	0.42929858904628254	0.8327383373067601
Functional API	0.013688536269600114	0.41624372973795126	0.8412600804260673

6. Conclusion

Based on my described work I will now conclude on the sub-questions.

6.1. What should the model predict?

From a business point of view the *revenue* is the clear winner. In many cases the point of releasing a movie is to earn some money. This is the reason why production companies agree to fund the productions and the reason why movies are hyped in the media through advertising. But is a movie successful if many watched it, maybe due to great advertising, but people actually disliked it and gave it bad ratings? All in all it depends on the goal in making the movie and rarely is it only one or the other. But if the objective was to release a movie that would be well loved and treasured by many, then using the *vote_average* as the label could potentially be a great way to go - especially since it will be able to utilize almost all the data in the dataset. This is something that I will be looking into in preparation for the oral exam.

6.2. What model should be used to predict success?

When using revenue as the label the best model to use can clearly be seen in the provided results. The model that performed the best was the Functional API. Rounded up the mean absolute error score finished at 0.0137, in percentage at 0.416, and an R2 score at 0.833. This is just slightly better than the Random Forest scores.
However, running both models several times does provide different results, and sometimes Random Forest even performs better than the Functional API model. So, when determining which model to use the answer would be either one of them. It could be that with more fine tuning and training the results might differ enough to conclude one better than the other, but as it is now, with the parameters given, the models are somewhat equal.

6.3. What attributes seem to be most relevant?

When making a movie it seems that casting the right actors is important if the movie has to earn a lot of money or be rated highly. This reflects “reality” in the sense that well loved and well known actors will have a higher value and salary. Combining the actors with the voting averages and revenue averages of the movies they had participated in showed these exact values’ correlation to the revenue and voting average of a film.
The budget also showed its significance in a movie's success, as expected. This is not surprising as the budget is what's paying for cast, crew, advertising, props and equipment; everything that’s contributing to the making of a movie.

6.4. Can the model be used by filmmakers?

Looking at just the MAE value and the R2 score it seemed that the models actually didn't perform too bad, but when comparing the MAE and the MAE in percentage to the revenue mean it's clear that they actually were about 50% off in their calculations. The models would definitely not be usable by filmmakers to predict or make a successful film. The data is based on movies spanding over many years and trends change, that's what makes them trendy, which might be the reason that they're seemingly unpredictable.

7. Reflection

7.1. The questions

The questions still stand and seem equally as relevant now. They were not too specific or restrictive, but simple and basic questions one has to ask when creating a model - which led me to have more freedom when answering them.

7.2. Method

I didn't really do anything that I hadn't mentioned under Method. But I did find myself researching more than I had expected and running into more problems than anticipated. Especially converting the JSON arrays gave me a lot of trouble due to mistakes in the data.
In the beginning I thought that I could mostly just follow what the book said about the topics I needed, but I had to resort to alternative sources for almost everything. Both to verify how to do things, but also because the book was insufficient for my specific needs or didn't handle the data / models the way I wanted to.

7.3. Plan

My plan was in general very fitting. I found that I mostly followed the process in the plan and it was very helpful writing all my intentions down, so I had a better overview throughout the whole project. But I was quite optimistic when thinking that I could write on my synopsis everyday. It's difficult to write about something when you don't know if what you did is correct or will work. A lot of the times this would only be revealed days or weeks later. For this reason I kept revisiting earlier work to change it or re-testing it. In the end I ended up writing about my process and findings the last two weeks. Although I did write a log book for every day and saved screenshots of anything important for comparisons if need be.

All in all I feel that I did well when preparing the project and I learned a lot about what's important and useful during a process like this. It has definitely been an altogether successful process.

8. Bibliography

Aurélien Géron *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd edition, O'Reilly 2019

Compulsory textbook for this semester. Used for examples of all models and Grid Search.

Rounak Banik *The Movies Dataset*, Kaggle.com, https://www.kaggle.com/rounakbanik/the-movies-dataset

Used for the dataset.

Pavel Bazin *Linear Regression: Implementation, Hyperparameters and their Optimizations* http://pavelbazin.com/post/linear-regression-hyperparameters/

Used example of how to make a Batch Gradient Descent test function.