AniFame

An anime popularity predictor

Lab Group: BCF3 Group 4
Toh Jing Qiang (U2121442H)
Toh Jing Hua (2121032L)
Xu YinFeng (U2121162B)

01.

Motivation

What drove us to embark on
a journey into anime analysis

Motivation:
High cost of producing anime

$2 million

"an average 13-episode anime season costs around 250 million yen (or $2 million)" (Eric, 2015)

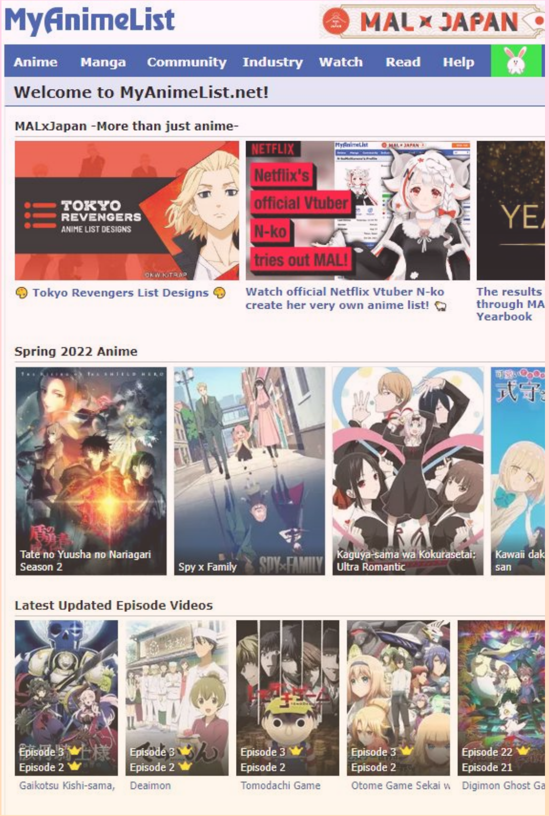
Motivation:
High cost of producing anime

Maximise Profits

Dataset used

MyAnimeList.net API

- Animes from 2000 to 2021
- Scrap and clean for EDA & ML



Presentation Outline

1. **Motivation**
   a. Problem Definition
   b. Dataset
2. **Setting the Stage**
   a. Data Collection
   b. Data Cleaning and Preprocessing
   c. Exploratory Data Analysis & Visualization
   d. Data-driven recommendations
3. **Core Analysis**
   a. Machine Learning
   b. Regression
   c. Classification
4. **Project Outcomes**
   a. Outcomes
   b. Interesting Things to Note
   c. More Data-driven Insights + Recommendations
   d. Conclusion
   e. Learning Points

02.

Setting the stage

EDA, data collection and
data preparation

2.1

Data Collection

[2.1 Data Collection]

Animes from 2000-2021 (100/season)

```python
def get_anime_season(year, season):
    # fetch 250 animes from a particular {season} of a particular {year}
    response = requests.get(f'https://api.myanimelist.net/v2/anime/season/{year}/{season}?limit=100',
                            headers={'X-MAL-CLIENT-ID': '6114d00ca681b7701d1e15fe11a4987e'})
    print(f'Status ({year}/{season})', response.status_code)


    for anime in response.json()['data']:
        anime_id = anime['node']['id']

        # query for anime details
        response_details = requests.get(f'https://api.myanimelist.net/v2/anime/{anime_id}?fields=id,title,start_date,end_date,synopsis,mean,rank,popu
                            headers={'X-MAL-CLIENT-ID': '6114d00ca681b7701d1e15fe11a4987e'})

        # add anime details to list
        anime_list.append(response_details.json())

    print(f'({year}/{season}) done!')
    print('---')
```

	id	title	main_picture	start_date	end_date	synopsis	mean	rank	popularity	num_list_users	...	genres	num_episodes	start_season	broadcast
0	95	Turn A Gundam	{'medium': 'https://api-cdn.myanimelist.net/im...	1999-04-09	2000-04-14	It is the Correct Century, two millennia after...	7.71	1049	2892	40743	...	[{'id': 1, 'name': 'Action'}, {'id': 2, 'name'...	50	{'year': 1999, 'season': 'spring'}	{'day_of_the_week': 'friday', 'start_time': '1...

Function to scrap by year and season

2.2
Data Cleaning & Preprocessing

Missing values

```
# No synopsis information available
data_clean["synopsis"].fillna(value = "no_Synopsis", inplace = True)

# Anime still airing/ongoing
data_clean["end_date"].fillna(value = "airing", inplace = True)

# Anime not broadcasted, replace missing value with the same formet for easy sorting
data_clean["broadcast"].fillna(value = "{'day_of_the_week': 'NIL', 'start_time': 'NIL'}", inplace = True)

# Source not known
data_clean["source"].fillna(value = "unknown", inplace = True)

# Not rated
data_clean["rating"].fillna(value = "no_rating", inplace = True)

# No genre information
data_clean["genres"].fillna(value = "[{'id': -1, 'name': 'no_genre'}]", inplace = True)

# Animes that do not have enough user giving their scorings, so replace null with value -1
data_clean["mean"].fillna(value = "-1", inplace = True)

# Check null values after cleaning
data_clean.isnull().sum()
```

Filling in NaN values with domain-specific values

[2.2b Data Cleaning & Preprocessing]

JSON Manipulation

Converting columns to JSON and splitting into individual columns for manipulation and feature engineering

- start_season
- broadcast
- statistics
- studios
- genres

```python
# Splitting start_season column into individual year and season columns
def split_start_season(data_clean):
    # create NaN columns
    data_clean['start_season_year'] = np.nan
    data_clean['start_season_season'] = np.nan


    for row in range(0,len(data_clean)):
        if data_clean['start_season'][row] == float('NaN'):
            continue

        #convert from string to json
        start_season = (json.loads(data_clean['start_season'][row].replace("'", "\"")))
        year = start_season['year']
        season = start_season['season']

        data_clean['start_season_year'][row] = year
        data_clean['start_season_season'][row] = season


    # drop original column
    data_clean.drop(columns=['start_season'], inplace=True)


    return data_clean
```

```python
# Splitting statistics column into watching, completed, on hold, plan to watch and num of user columns

def split_statistics(data_clean):
    # create NaN columns
    data_clean['statistics_watching'] = np.nan
    data_clean['statistics_completed'] = np.nan
    data_clean['statistics_on_hold'] = np.nan
    data_clean['statistics_dropped'] = np.nan
    data_clean['statistics_plan_to_watch'] = np.nan
    data_clean['statistics_num_list_users'] = np.nan


    for row in range(0,len(data_clean)):
        # convert from string to json
        statistics = (json.loads(data_clean['statistics'][row].replace("'", "\"")))

        data_clean['statistics_watching'][row] = statistics['status']['watching']
        data_clean['statistics_completed'][row] = statistics['status']['completed']
        data_clean['statistics_on_hold'][row] = statistics['status']['on_hold']
        data_clean['statistics_dropped'][row] = statistics['status']['dropped']
        data_clean['statistics_plan_to_watch'][row] = statistics['status']['plan_to_watch']
        data_clean['statistics_num_list_users'][row] = statistics['num_list_users']


    # drop original column
    data_clean.drop(columns=['statistics'], inplace=True)


    return data_clean
```

```python
# Splitting broadcast column into individual day and time columns
def split_broadcast(data_clean):
    # create NaN columns
    data_clean['broadcast_day_of_the_week'] = np.nan
    data_clean['broadcast_start_time'] = np.nan


    for row in range(0,len(data_clean)):
        #convert from string to json
        broadcast = (json.loads(data_clean['broadcast'][row].replace("'", "\"")))


        data_clean['broadcast_day_of_the_week'][row] = broadcast['day_of_the_week']

        try:
            data_clean['broadcast_start_time'][row] = broadcast['start_time']
        except:
            data_clean['broadcast_start_time'][row] = 'NIL'


    # drop original column
    data_clean.drop(columns=['broadcast'], inplace=True)


    return data_clean
```

```python
# Convert genres into json format
def json_genres(data_clean):
    #Convert genres string to json
    for row in range(0, len(data_clean)):
        genres = json.loads(data_clean['genres'][row].replace("'", "\""))

        data_clean['genres'][row] = genres

    return data_clean
```

```python
# Convert studios into json format
def json_studios(data_clean):
    # Convert studios string to json
    for row in range(0, len(data_clean)):
        try:
            studios = (json.loads(data_clean['studios'][row].replace("'", "\"")))
        except:
            studios = (json.loads(data_clean['studios'][row].replace("'", "\"").replace("\"s", '\'s').replace('N\"', "N\'")))

        data_clean['studios'][row] = studios

    return data_clean
```

Functions to convert and splitting JSON columns

Feature Engineering

New Features Generated:

- **From 'broadcast':**
  - broadcast_day_of_the_week
  - broadcast_start_time
- **From 'start_season'**
  - start_season_year
  - start_season_season
- **From 'statistics'**
  - statistics_watching
  - statistics_completed
  - statistics_on_hold
  - statistics_dropped
  - statistics_plan_to_watch
  - statistics_num_list_users
  - **Aggregation:**
    - positive_viewership_fraction: statistics_watching + statistics_completed + statistics_plan_to_watch
    - negative_viewership_fraction: statistics_on_hold + statistics_dropped

Creating positive/negative viewership feature

```python
# Create percentage fraction positive/negative viewerships --> Range: [0, 1]
# *function to be called after split_statistics() function*

def get_pos_neg_viewership(anime, viewership_types_list):
    # single anime
    total_pos_neg_views = 0

    for viewership_type in viewership_types_list:
        total_pos_neg_views += data_clean[viewership_type][anime]

    return total_pos_neg_views

def create_viewership_fraction(data_clean):
    # create NaN columns
    data_clean['positive_viewership_fraction'] = np.nan
    data_clean['negative_viewership_fraction'] = np.nan

    positive_viewership = [
        'statistics_watching',
        'statistics_completed',
        'statistics_plan_to_watch'
    ]
    negative_viewership = [
        'statistics_on_hold',
        'statistics_dropped'
    ]

    for anime in range(0, len(data_clean)):
        total_views = data_clean['statistics_num_list_users'][anime]

        # calulating the total postive and total negative views respectively
        total_pos_views = get_pos_neg_viewership(anime, positive_viewership)
        total_neg_views = get_pos_neg_viewership(anime, negative_viewership)

        # calculate percentage fraction & create a new column
        data_clean['positive_viewership_fraction'][anime] = round(total_pos_views/total_views, 4)
        data_clean['negative_viewership_fraction'][anime] = round(total_neg_views/total_views, 4)

    return data_clean
```

Feature Engineering - 'success'

'success' (**1**: successful, **0**: not successful)

- Top 500 *rank*
- Top 500 *popularity*
- *mean* above 8.5
- *positive_viewership_fraction* above 0.975

```python
# create 'success' column
anime_df['success'] = np.nan

for row in range(len(anime_df)):
    success = (anime_df['rank'][row] <= 500 or
               anime_df['popularity'][row] <= 500 or
               anime_df['mean'][row] >= 8.5 or
               anime_df['positive_viewership_fraction'][row] >= 0.975)

    if success:
        anime_df['success'][row] = 1
    else:
        anime_df['success'][row] = 0
```

[2.2e Data Cleaning & Preprocessing]

Time Series - Genres

Using start_season_year and genres to create genre time series dataframe for analysis

```python
for row in range(len(genres_time_series_df)):
    skip = False

    single_year_row = {}

    start_season_year = genres_time_series_df['start_season_year'][row]

    # skip years earlier than 1999
    if start_season_year < 1999.0:
        continue

    # if start season year already exists in the dataframe, just add
    for year in new_genres_time_series_df['Start Season Year']:
        if start_season_year == year:
            # add to dataframe
            genre = genres_time_series_df['genre'][row]
            genre_count = genres_time_series_df['count'][row]

            new_genres_time_series_df.loc[new_genres_time_series_df['Start Season Year'] == start_seas

            skip = True
            break

    if skip:
        continue

    single_year_row['Start Season Year'] = [start_season_year]

    for genre in genres_list:
        # add to dictionary the start season year and count
        if genre == genres_time_series_df['genre'][row]:
            single_year_row[genre] = [genres_time_series_df['count'][row]]
        else:
            single_year_row[genre] = [0]

    # add to dataframe
    new_genres_time_series_df = new_genres_time_series_df.append(pd.DataFrame(single_year_row), ignore

new_genres_time_series_df
```

	Start Season Year	School	Suspense	Mystery	Adventure	Slice of Life	Sports	Martial Arts	Space	Comedy	...	Shounen	Game	Shoujo	Sci-Fi
0	1999.0	3	1	20	127	35	0	0	6	167	...	103	0	1	38
1	2000.0	14	0	18	144	12	10	4	12	172	...	88	26	22	76
2	2001.0	34	0	14	110	36	41	8	15	161	...	95	6	25	88
3	2002.0	40	0	27	149	51	23	25	19	234	...	111	21	32	127
4	2003.0	18	0	20	148	26	26	6	8	168	...	110	12	20	126
5	2004.0	31	13	34	155	23	23	9	20	223	...	169	23	34	127
6	2005.0	24	3	24	102	30	23	5	9	202	...	139	16	32	124
7	2006.0	33	6	31	112	23	6	13	8	212	...	108	15	33	82
8	2007.0	41	8	35	112	24	19	48	10	166	...	86	8	23	58
9	2008.0	54	11	25	81	35	17	7	8	167	...	79	21	31	62
10	2009.0	49	13	31	87	60	22	15	12	206	...	77	6	21	49
11	2010.0	42	5	19	67	38	30	9	14	194	...	61	17	21	46
12	2011.0	53	10	26	68	45	15	11	2	155	...	97	16	32	43
13	2012.0	75	6	28	44	74	26	9	12	159	...	72	17	23	53
14	2013.0	69	4	18	51	59	29	5	10	157	...	67	23	21	27
15	2014.0	80	5	23	71	66	18	12	13	174	...	83	22	37	49
16	2015.0	70	4	23	56	64	22	13	14	158	...	81	19	27	47
17	2016.0	77	2	29	47	90	25	4	11	151	...	61	18	29	33
18	2017.0	50	3	23	41	56	28	5	4	128	...	60	27	14	31
19	2018.0	41	6	22	60	88	20	19	5	142	...	49	22	24	29
20	2019.0	52	5	18	58	60	17	17	10	133	...	47	21	21	53
21	2020.0	38	5	19	80	51	14	26	0	120	...	29	14	13	31
22	2021.0	38	8	24	64	59	11	25	8	110	...	52	7	13	38

23 rows × 41 columns

One Hot Encoding

One hot encoding of categorical variables:

- media_type
- source
- rating
- start_season_season
- start_season_year
- status
- nsfw
- genres
- studios

```
# Import the encoder from sklearn
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder()

# OneHotEncoding of categorical predictors (not the response)
cat_variables = [
    'media_type', 'source', 'rating', 'start_season_season',
    'start_season_year', 'status', 'nsfw'
] + [f"genre-{i}" for i in genres_expanded.columns] + [f"studio-{i}" for i in studios_expanded.columns]
anime_cat = anime_expanded_df[cat_variables]

ohe.fit(anime_cat)
anime_cat_ohe = pd.DataFrame(ohe.transform(anime_cat).toarray(),
                             columns=ohe.get_feature_names(anime_cat.columns))

# Check the encoded variables
anime_cat_ohe.info()
```

```
media_type_movie
media_type_music
media_type_ona
media_type_ova
media_type_special
media_type_tv
source_4_koma_manga
source_book
source_card_game
source_digital_manga
source_game
source_light_novel
source_manga
source_mixed_media
source_music
source_novel
source_original
source_other
source_picture_book
source_radio
source_unknown
source_visual_novel
source_web_manga
source_web_novel
rating_g
rating_no_rating
rating_pg
rating_pg_13
rating_r
rating_r+
start_season_season_fall
start_season_season_spring
start_season_season_summer
```

2.3
Exploratory Data Analysis & Visualization

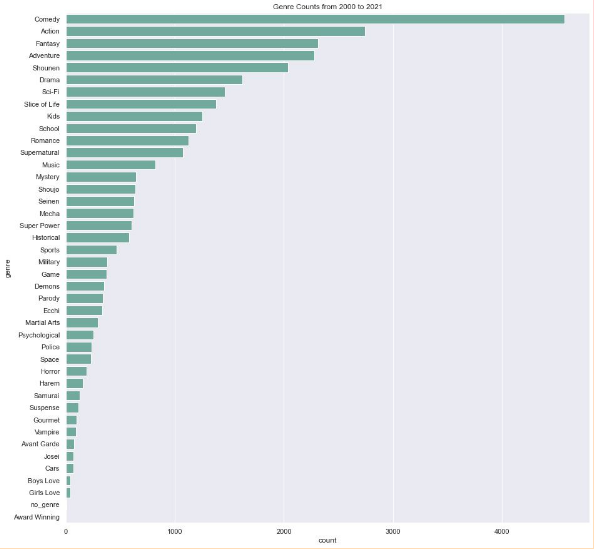
Genres

Top 5 genres from 2000 to 2021:

- Comedy, action, fantasy, adventure and shounen

Genres trend from 2000 to 2021:

- **[Decreasing Trend]** ‘Shounen’, ‘Comedy’ and ‘Adventure’
- **[Increasing Trend]** ‘Slice of Life’, and ‘Music’



[2.3a Exploratory Data Analysis & Visualization]

Genres

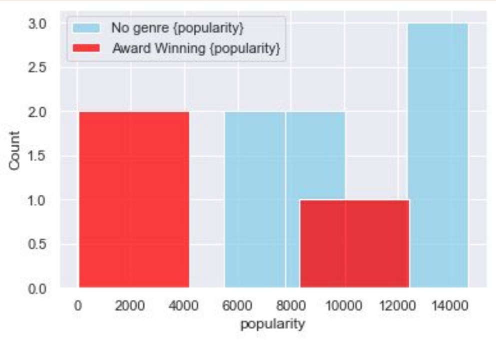
- It is surprising to see that 'Shounen', 'Comedy' and 'Adventure' have a decreasing trend

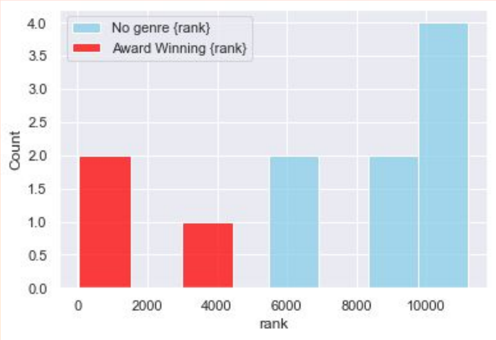
Award Winning vs No Genre

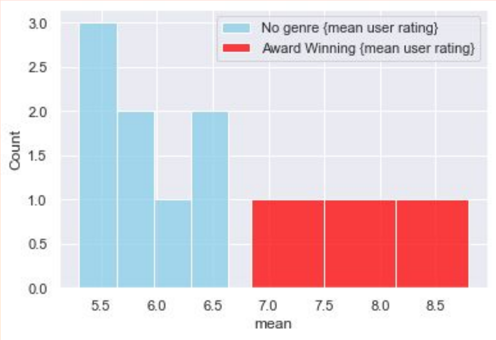
Comparing animes with ‘Award Winning’ and ‘no_genre’ genres

‘Award Winning’ animes:

- Higher popularity
- Higher ranked
- Higher ratings





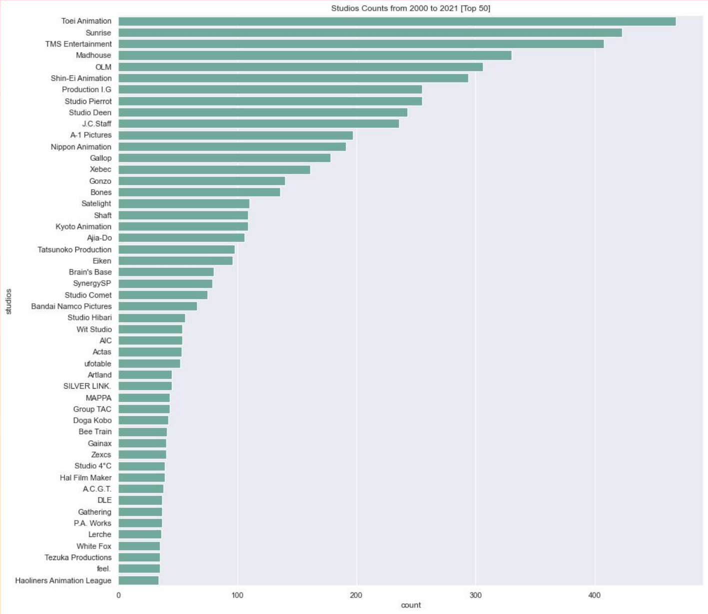


[2.3c Exploratory Data Analysis & Visualization]

Studios

Top 5 anime studios from 2000 to 2021:

- Toei Animation, Sunrise, TMS Entertainment, Madhouse, OLM

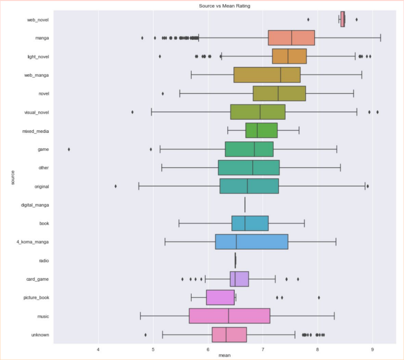


Mean rating vs various features

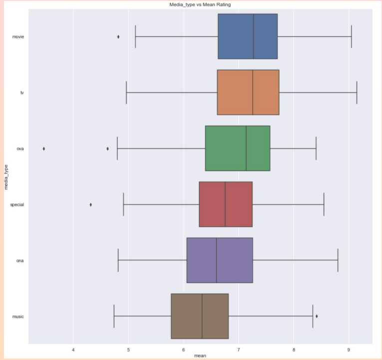
Mean rating compared with:

- 'source'
- 'media_type'
- 'rating'
- 'genres'
- 'studios'

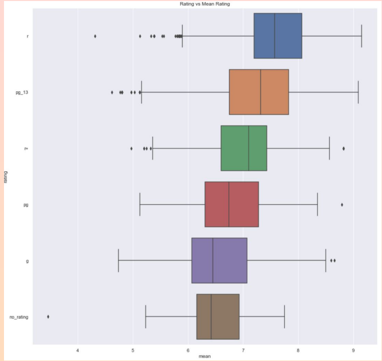
Source vs Mean



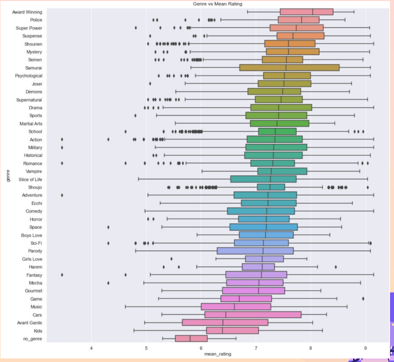
Media Type vs Mean



Rating vs Mean



Genres vs Mean

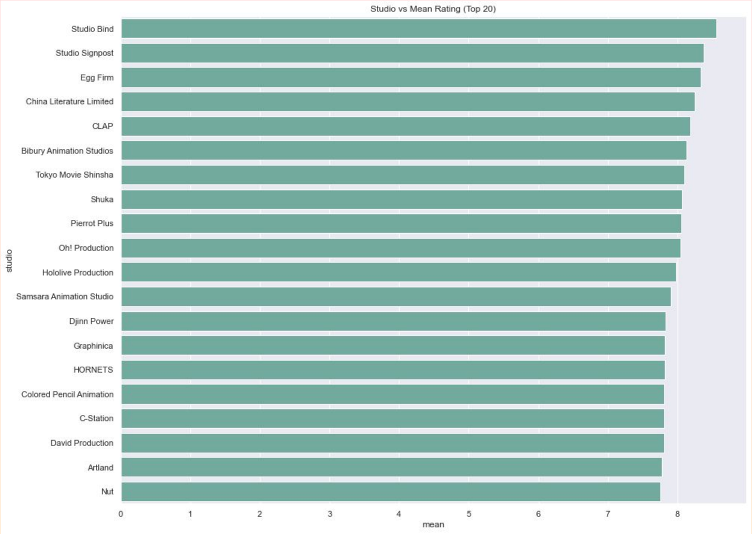


[2.3d Exploratory Data Analysis & Visualization]

Mean rating vs Studios

Studio of the anime vs mean rating of the anime:

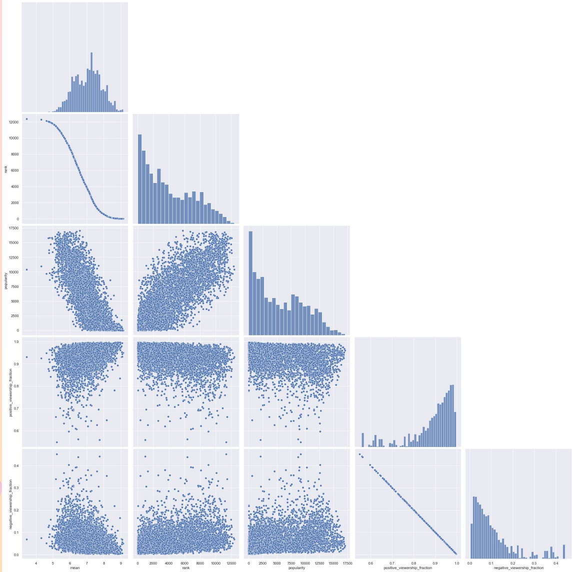
- **Quality** is better than quantity
- The top 5 most common studios are not seen in top 20 studios with highest mean ratings

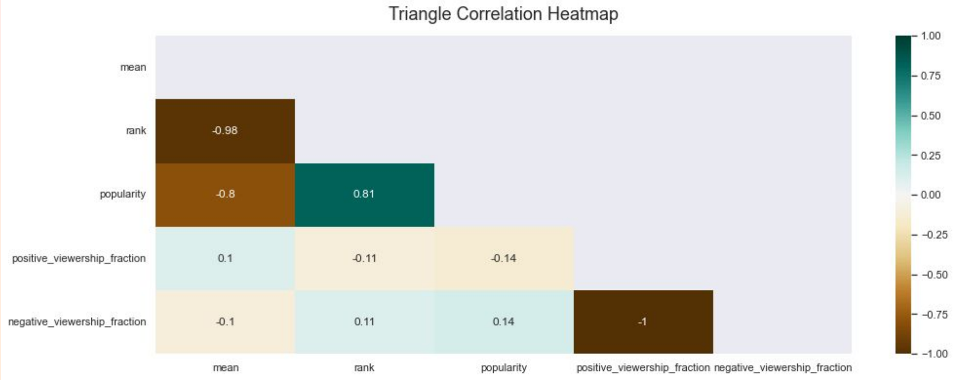


Multivariate Relationships

Relationship between mean, rank, popularity, positive/negative viewership:

- mean, rank and popularity are **correlated**
- negative/positive viewership have **no significant correlation** with mean, rank, popularity





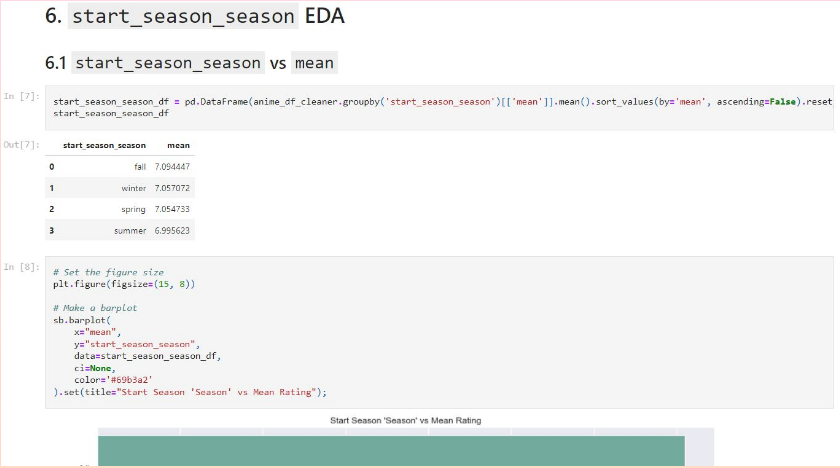
[2.3f Exploratory Data Analysis & Visualization]

More EDA

More EDA found in Jupyter notebook:

- num_episodes
- average_episode_duration
- start_season_season

- Previous EDA in details



6. `start_season_season` EDA

6.1 `start_season_season` vs `mean`

In [7]:

```
start_season_season_df = pd.DataFrame(anime_df_cleaner.groupby('start_season_season')[['mean']].mean().sort_values(by='mean', ascending=False).reset
start_season_season_df
```

Out[7]:

	start_season_season	mean
0	fall	7.094447
1	winter	7.057072
2	spring	7.054733
3	summer	6.995623

In [8]:

```
# Set the figure size
plt.figure(figsize=(15, 8))

# Make a barplot
sb.barplot(
    x="mean",
    y="start_season_season",
    data=start_season_season_df,
    ci=None,
    color='#69b3a2'
).set(title="Start Season 'Season' vs Mean Rating");
```

Data-driven recommendations

Data-driven Recommendations:

- Studios should
    a. Focus on quality instead of quantity of anime
    b. Broadcast anime regardless of the season
    c. Not focus on producing anime that generate more positive views through fan-services

03.

Core Analysis

Regression + Classification

Machine Learning

Classification

- Determine **probability** of success of an anime **(Yes/No)**

Regression

- **Predict** mean rating
- High mean rating == anime well-received (positive correlation with ranking & popularity)

Objective

- Studios can predict the mean rating and classify the probability of success of the anime before production
- Maximizing profits from viewership, events and merchandise sales from pre-production anime fine-tuning

[Regression Models]

Regression

Goal:

- **Estimate** ‘mean’ rating of an anime based on the features of animes before they are produced

Models:

- Linear Regression
- Lasso Regression
- **Ridge Regression** [Best]

Best regression model

Ranking of regression models:

1. Ridge regression (`~0.7`)
2. Lasso regression (`~0.6`)
3. Linear regression (`~0.4`)

Why ridge regression worked better:

- We had an enormous amount of variables in our dataset (`900+ variables`) and using normal linear regression to fit all the variables may result in overfitting
- Ridge regression helps minimise overfitting by regularising the coefficients. This causes some coefficients to be near `0` .
- This helps us to select relevant features by making the coefficients of irrelevant features to be almost 0.
- Hence, ridge regression reduces overfitting and increases the performance of the model.

Why lasso regression performed slightly worse than ridge regression:

- Lasso regression made the coefficients of some variables `0` .
- This could have reduced the accuracy as the variables might have had some impact on the prediction as well
- Hence, it performed slightly worse than ridge regression

[Classification Models + Performance Metrics]

Classification

Goal:

- **Classify** future success based on features of animes before they are produced

How:

- Predicting the probability of '1' in the **'success' feature**

Models:

- LinearSVC
- Decision Trees
- **Random Forest** [Best]

Performance Metrics:

- K-fold cross validation (K = 5)
  - TPR, TNR, Confusion Matrix
  - Precision, Recall (TPR), F1 score
  - ROC AUC Score
  - Out-of-bag (oob) score for random forest models
  - Performance consistency (standard deviation)

In [22]:

```python
def model_performance(random_forest, X_train, X_test, y_train, y_test):
    # Import libraries
    from sklearn.model_selection import cross_val_predict
    from sklearn.model_selection import cross_val_score

    # K-Fold Cross Validation
    y_train_pred = cross_val_predict(random_forest, X_train, y_train, cv=5)
    y_test_pred = cross_val_predict(random_forest, X_test, y_test, cv=5)

    train_scores = cross_val_score(random_forest, X_train, y_train, cv=5, scoring = "accuracy")
    test_scores = cross_val_score(random_forest, X_test, y_test, cv=5, scoring = "accuracy")

    # Performance metrics
    #confusion_matrix_TPR_TNR(y_train, y_test, y_train_pred, y_test_pred, train_scores, test_scores)



    print("-> Train Dataset")
    confusion_matrix_TPR_TNR(y_train, y_train_pred, train_scores, "Train")
    get_precision_recall(y_train, y_train_pred)
    get_f1_score(y_train, y_train_pred)

    print("\n\n-----\n-> Test Dataset")
    confusion_matrix_TPR_TNR(y_test, y_test_pred, test_scores, "Test")
    get_precision_recall(y_test, y_test_pred)
    get_f1_score(y_test, y_test_pred)
    ROC_AUC(random_forest, X_test, y_test, "Test")
    print('-----\n')

    try:
        get_oob_score(random_forest)
    except:
        pass
```

[Classification]

LinearSVC

Reason for trying:

- Large dataset with many rows and features

Performance:

- **[Poor]** Very low classification accuracy (~0.6), true positive rate, precision, recall, and f1_score (~0.2) for both train and test dataset

Reason for performance:

- LinearSVC more suited for text classification instead of categorical and continuous dataset

```
-> Train Dataset
Goodness of Fit of Model (Train Dataset)
Classification Accuracy: 0.5658134238815472
True Positive Rate: 0.4711246200607926
True Negative Rate: 0.7542561065877128

Precision: 0.4711246200607926
Recall: 0.18925518925518925
f1_score: 0.2700348432055749


-----
-> Test Dataset
Goodness of Fit of Model (Test Dataset)
Classification Accuracy: 0.6525811471765229
True Positive Rate: 0.4298642533936652
True Negative Rate: 0.7304457527333894

Precision: 0.4298642533936652
Recall: 0.12907608695652173
f1_score: 0.19853709508881923
-----
```

[Classification]

Decision Tree

Reason for trying:

- Categorical and continuous dataset

Performance:

- **[Decent]**
- ⬆️ Classification accuracy (~0.8)
- ⬆️ ROC AUC Score (~0.8)
- ⬆️ True positive rate, precision, recall, and F1 score

Reason for performance:

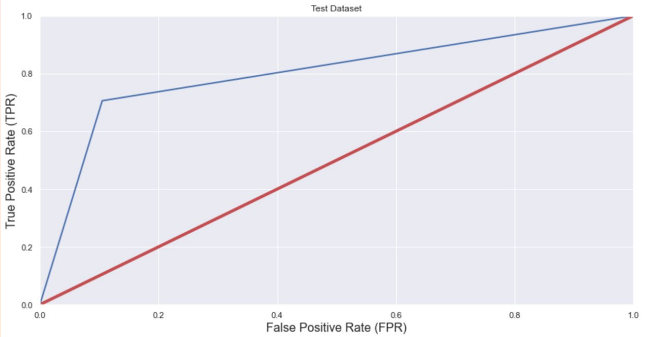
- Categorical and continuous dataset suited

```
-> Train Dataset
Goodness of Fit of Model (Train Dataset)
Classification Accuracy: 0.8330556757242089
True Positive Rate: 0.6963657678780774
True Negative Rate: 0.8939393939393939

Precision: 0.6963657678780774
Recall: 0.72
f1_score: 0.7079856972586411


-----
-> Test Dataset
Goodness of Fit of Model (Test Dataset)
Classification Accuracy: 0.810316436934934
True Positive Rate: 0.6642547033285094
True Negative Rate: 0.8611111111111112

Precision: 0.6642547033285094
Recall: 0.6339779005524862
f1_score: 0.6487632508833923
```



ROC-AUC-Score: 0.8003672191528546

[Classification]

Random Forest V1

Reason for trying:

- Ensemble of decision trees (Many trees built)

Performance:

- **[Good]**
- ⬆️ Classification accuracy, TPR, F1 Score
- ⬆️ ROC AUC Score (~0.8 to ~ 0.94)

Reason for performance:

- Random Forest builds multiple decision trees and merge them together to get a more accurate and stable prediction
- Random Forest prevent overfitting on datasets

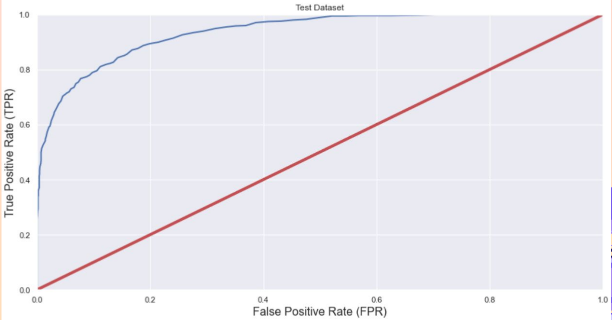
```
-> Train Dataset
Goodness of Fit of Model (Train Dataset)
Classification Accuracy: 0.8812277064474792
True Positive Rate: 0.8696356275303644
True Negative Rate: 0.8848145846281334

Precision: 0.8696356275303644
Recall: 0.6588957055214724
f1_score: 0.7497382198952879


-----
-> Test Dataset
Goodness of Fit of Model (Test Dataset)
Classification Accuracy: 0.8464784348599377
True Positive Rate: 0.8163265306122449
True Negative Rate: 0.8524271844660194

Precision: 0.8163265306122449
Recall: 0.5913978494623656
f1_score: 0.6858924395947
```

```
Standard Deviation: 0.008283311261719373
```




```
ROC-AUC-Score: 0.9367265165347941
-----

oob score: 88.4 %
```

[Classification **Feature Importance**]

Random Forest V2

Random Forest Improvement:

- **[Feature Importance]**
- Removing 600+ features with ‘0’ importance

Performance:

- **[Great]**
- ⬆️ Classification accuracy, TPR, TNR, F1 Score
- ⬆️ ROC AUC Score
- ⬆️ Performance speed & consistency (s.d. 0.00409)
- ↔ Oob score

Reason for performance:

- Dimensionality of the model is reduced → ⬆️ Model speed & Performance since only important features are considered.
- Prevents overfitting, however performance only increase slightly as random forest models tend not to overfit

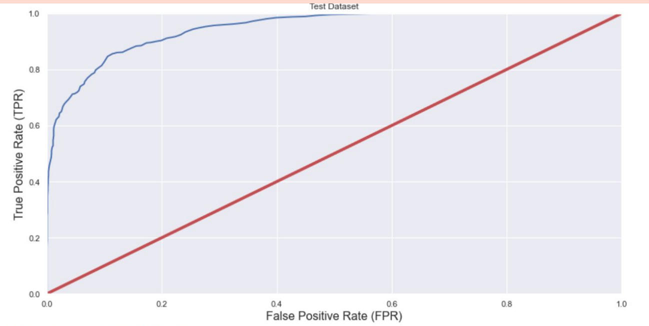
```
-> Train Dataset
Goodness of Fit of Model (Train Dataset)
Classification Accuracy: 0.8785870343011218
True Positive Rate: 0.8598425196850393
True Negative Rate: 0.8831385642737897

Precision: 0.8598425196850393
Recall: 0.6610169491525424
f1_score: 0.7474332648870635


-----
-> Test Dataset
Goodness of Fit of Model (Test Dataset)
Classification Accuracy: 0.863028753520829
True Positive Rate: 0.8403041825095057
True Negative Rate: 0.8649300530631935

Precision: 0.8403041825095057
Recall: 0.6121883656509696
f1_score: 0.7083333333333334
```

```
Standard Deviation: 0.004094974463354793
```




```
ROC-AUC-Score: 0.9467519041554198
-----

oob score: 88.16 %
```

[Classification **Feature Importance**]

Random Forest V3

Random Forest Change:

- **[Feature Importance]**
- Top 50 important features

Performance:

- **[Good but decreased performance]**
- ⬇️ Classification accuracy, precision, recall, F1 Score
- ⬇️ ROC AUC Score
- ⬇️ Performance consistency
- ⬇️ Oob score
- ⬆️ Performance speed

Reason for performance:

- Only 50 out of about 250+ important features were considered before splitting a node
- Reducing large number of features reduces the performance but increases the speed

```
-> Train Dataset
Goodness of Fit of Model (Train Dataset)
Classification Accuracy: 0.862422627258604
True Positive Rate: 0.7997021593447505
True Negative Rate: 0.879423606696334

Precision: 0.7997021593447505
Recall: 0.6536822884966524
f1_score: 0.7193569993302077


-----
-> Test Dataset
Goodness of Fit of Model (Test Dataset)
Classification Accuracy: 0.841476211649622
True Positive Rate: 0.7504105090311987
True Negative Rate: 0.8623115577889447

Precision: 0.7504105090311987
Recall: 0.625170998632011
f1_score: 0.682089552238806
```

```
Standard Deviation: 0.0066373903265731235
```


```
ROC-AUC-Score: 0.935843656719697
-----
oob score: 86.7 %
```

[Classification **Hyperparameter Tuning**]

Random Forest V4

Random Forest Improvement:

- **[Hyperparameter Tuning]**
- 'criterion': **'entropy'**
- 'n_estimators': **700**

Performance:

- **[Excellent]**
- ⬆️ Classification accuracy, precision, recall, F1 Score, ROC AUC Score, Oob score, and performance consistency
- ↔ TPR, TNR
- Performance speed between v1 & v2

Reason for performance:

- Entropy 'criterion':
  - Measures the *disorder of features*
  - Dataset is more suited for using entropy
- 'n_estimators':
  - *Number of trees* built before taking the maximum voting or averages of prediction
  - Having a value of 700 over the default 100 is used as building more trees leads to better performance

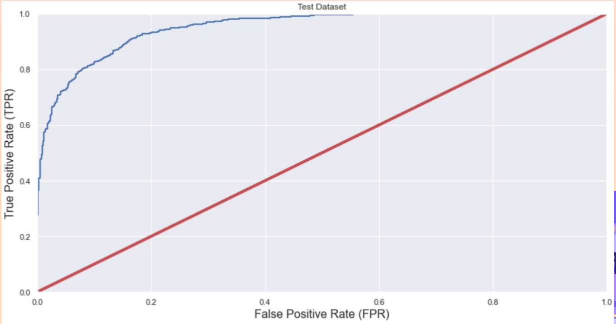
```
-> Train Dataset
Goodness of Fit of Model (Train Dataset)
Classification Accuracy: 0.8792495490274502
True Positive Rate: 0.861244019138756
True Negative Rate: 0.8839434276206323

Precision: 0.861244019138756
Recall: 0.6593406593406593
f1_score: 0.7468879668049794


-----
-> Test Dataset
Goodness of Fit of Model (Test Dataset)
Classification Accuracy: 0.8518734252260265
True Positive Rate: 0.8305084745762712
True Negative Rate: 0.8573500967117988

Precision: 0.8305084745762712
Recall: 0.5991847826086957
f1_score: 0.6961325966850828
```

```
Standard Deviation: 0.004661087566371556
```




```
ROC-AUC-Score: 0.9502343987024202
-----

oob score: 88.8 %
```

[Classification]

Model Comparison

Models Built:

- LinearSVC
- Decision Tree
- Random Forest V1
- Random Forest V2
- Random Forest V3
- Random Forest V4

Model to Use:

- **Random Forest V4**
  - Classification accuracy of **89%** on the test dataset
  - Excellent performance consistency and performance speed
  - Great performance metrics

	Model	Score
0	Random Forest V4	89.15
1	Random Forest V2	88.76
2	Random Forest V1	88.00
3	Random Forest V3	87.88
4	Decision Tree	84.22
5	Support Vector Machines	71.91

04.

Project Outcomes

Outcome, Insights, Recommendations, Learning Points

[Project outcomes]

Outcomes

Important Features:

- 'average_episode_duration'
- 'num_episodes'
- 'source_manga'
- 'media_type_movie'
- 'rating_pg_13'

Classification:

- Classify animes success probability with high accuracy of **89%** [Random Forest V4]

Regression:

- Estimate 'mean' rating of animes reliably with about **0.7 R^2** [Ridge Regression]

Solving original Problem:

- Studios can **fine-tune** the anime before production and **maximize** their **profits** after production, ensuring their survivability in the industry

Interesting Things to Note

- Shounen, Comedy, and Adventure genres have a decreasing trend since they are among the top 5 genres commonly seen. Instead, Slice of Life and Music genres have an increasing trend. Thus, there is a **shift in the genres trend** that studios can take note of.
- **Quality > quantity** for increasing mean rating and thus profits.
- Random forest models have determined that over 70% of total number of features are not important.
    - This shows that **feature engineering** and **selection** is important in building machine learning models.

[Project Outcomes]

Data-driven Insights + Recommendations

More Insights:

- Important features that determine the success of an anime
  - ‘average_episode_duration’
  - ‘num_episodes’
  - ‘source_manga’
  - ‘media_type_movie’
  - ‘rating_pg_13’

More Recommendations:

- Studios should try to produce anime that originates from manga, has a pg_13 rating, and as a movie, which have a low number of episodes and long average episode duration.
- Movie franchises will likely be more successful than just regular anime. Therefore, studios should produce **anime movie franchises** too.

Data-driven Recommendations: [From EDA presented previously]

- Studios should
  a. Focus on quality instead of quantity of anime
  b. Broadcast anime regardless of the season
  c. Not focus on producing anime that generate more positive views through fan-services

[Project Outcomes]

Conclusion

Anime fine-tuning &
Maximize studios' profits

[Project Outcomes]

Learning Points

Data collection:

- Scraping data using API calls

Data cleaning and preprocessing:

- Feature Engineering & Feature generation
- JSON manipulation techniques
- Generating time-series data

EDA & Visualization:

- Visualization plots with large number of datapoints
  - By reducing the data point size,
  - By reducing the opacity of data points, or
  - By introducing random sampling
- ‘genres’ time-series EDA

Machine Learning:

- Machine Learning Models:
  - Ridge Regression, Lasso Regression, Random Forest, LinearSVC
- Classification Performance Metrics:
  - F-score (Precision & Recall), out-of-bag (obb) score, ROC AUC score

Thank You!