In the second part of the notebook, we will focus on constructing a regression model.

Table of Contents


```
pip install scikeras
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/pul
Collecting scikeras
  Downloading scikeras-0.10.0-py3-none-any.whl (27 kB)
Requirement already satisfied: packaging>=0.21 in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: scikit-learn>=1.0.0 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-
Installing collected packages: scikeras
Successfully installed scikeras-0.10.0
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots

# Plotly

import plotly.graph_objects as go
```

```
# Sklearn
import seaborn as sns
import sklearn
from tqdm.notebook import tqdm
from sklearn.ensemble import RandomForestRegressor
import statsmodels.api as sm
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import AdaBoostRegressor
from xgboost import XGBRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV

from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.compose import ColumnTransformer

# Deep learning
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
from keras.callbacks import ReduceLROnPlateau, EarlyStopping
from scikeras.wrappers import KerasRegressor
from keras.models import load_model
import warnings

# Suppress all warnings
warnings.filterwarnings("ignore")
```

Gathering data:

In this notebook, we have two data frames. One includes movie ID, title, cast, and crew, while the second has additional features like genres, budget, and original language. We will merge these data frames since they both contain useful information for predicting movie popularity. By combining the data, we can utilize a broader set of predictors to enhance the accuracy of our popularity predictions.

```
# Uploading and viewing the data
tmdb_5000_cred = pd.read_csv(r'tmdb_5000_credits.csv', index_col=False)
tmdb_5000_cred.head()
```

	movie_id	title	cast	
0	19995	Avatar	[{"cast_id": 242, "character": "Jake Sully", "...	[{"credi
1	285	Pirates of the Caribbean: At World's End	[{"cast_id": 4, "character": "Captain Jack Spa...	[{"cred
2	206647	Spectre	[{"cast_id": 1, "character": "James Bond", "cr...	[{"credit
3	49026	The Dark Knight Rises	[{"cast_id": 2, "character": "Bruce Wayne / Ba...	[{"crec
4	49529	John Carter	[{"cast_id": 5, "character": "John Carter", "c...	[{"cred

```
# Uploading and viewing the data
tmdb_5000_cred.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4803 entries, 0 to 4802
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   movie_id  4803 non-null   int64 
 1   title     4803 non-null   object
 2   cast      4803 non-null   object
 3   crew      4803 non-null   object
dtypes: int64(1), object(3)
memory usage: 150.2+ KB
```

```
# Uploading and viewing the data
tmdb_5000_mov = pd.read_csv(r'tmdb_5000_movies.csv')
tmdb_5000_mov.head()
```

	budget	genres	homepage	id	keywords	original_l
0	237000000	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	http://www.avatarmovie.com/	19995	[{"id": 1463, "name": "culture clash"}, {"id":...	
1	300000000	[{"id": 12, "name": "Adventure"}, {"id": 14, "...	http://disney.go.com/disneypictures/pirates/	285	[{"id": 270, "name": "ocean"}, {"id": 726, "na...	
		[{"id": 28, "name":			[{"id": 470, "name":	

```
# Checking the details of the data
tmdb_5000_mov.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4803 entries, 0 to 4802
Data columns (total 20 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   budget                4803 non-null   int64
 1   genres                4803 non-null   object
 2   homepage              1712 non-null   object
 3   id                    4803 non-null   int64
 4   keywords              4803 non-null   object
 5   original_language     4803 non-null   object
 6   original_title        4803 non-null   object
 7   overview              4800 non-null   object
 8   popularity            4803 non-null   float64
 9   production_companies  4803 non-null   object
 10  production_countries  4803 non-null   object
 11  release_date          4802 non-null   object
 12  revenue               4803 non-null   int64
 13  runtime               4801 non-null   float64
 14  spoken_languages      4803 non-null   object
 15  status                4803 non-null   object
 16  tagline               3959 non-null   object
 17  title                 4803 non-null   object
 18  vote_average          4803 non-null   float64
 19  vote_count            4803 non-null   int64
dtypes: float64(3), int64(4), object(13)
memory usage: 750.6+ KB
```

Merging the data

```
# Merging the two data sets
tmdb_5000_cred.columns = ['id','tittle','cast','crew']
tmdb_5000_mov = tmdb_5000_mov.merge(tmdb_5000_cred,on='id')
```

```
# View more details
tmdb_5000_mov.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4803 entries, 0 to 4802
Data columns (total 23 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   budget                4803 non-null   int64  
 1   genres                4803 non-null   object 
 2   homepage              1712 non-null   object 
 3   id                    4803 non-null   int64  
 4   keywords              4803 non-null   object 
 5   original_language     4803 non-null   object 
 6   original_title        4803 non-null   object 
 7   overview              4800 non-null   object 
 8   popularity            4803 non-null   float64
 9   production_companies  4803 non-null   object 
 10  production_countries  4803 non-null   object 
 11  release_date          4802 non-null   object 
 12  revenue               4803 non-null   int64  
 13  runtime               4801 non-null   float64
 14  spoken_languages      4803 non-null   object 
 15  status                4803 non-null   object 
 16  tagline               3959 non-null   object 
 17  title                 4803 non-null   object 
 18  vote_average          4803 non-null   float64
 19  vote_count            4803 non-null   int64  
 20  tittle                4803 non-null   object 
 21  cast                  4803 non-null   object 
 22  crew                  4803 non-null   object 
dtypes: float64(3), int64(4), object(16)
memory usage: 900.6+ KB
```

The value count in the "tagline" and "homepage" columns shows null values which we will handle next.

Cleaning Data

Handling null values

```
# Count null values in each column
null_counts = tmdb_5000_mov.isnull().sum()

# Print the null value counts for each column
print(null_counts)
```

```
budget                     0
genres                     0
homepage                3091
id                         0
keywords                   0
original_language          0
```

```
original_title            0
overview                  3
popularity                0
production_companies      0
production_countries      0
release_date              1
revenue                   0
runtime                   2
spoken_languages          0
status                    0
tagline                 844
title                     0
vote_average              0
vote_count                0
tittle                    0
cast                      0
crew                      0
dtype: int64
```

This is how we will handle the null values:

- homepage' has 3091 null values. - we will remove the column
- 'overview' has 3 null values. - we will remove the null
- 'release_date' has 1 null value. - we will remove the null values
- 'runtime' has 2 null values. - we will remove the null
- 'tagline' has 844 null values. - We will remove the column

```
# Dropping columns with high null values (homepage and tagline)
tmdb_5000_mov.drop(['homepage', 'tagline'], axis=1, inplace=True)
```

```
# Removing the remaining rows with null values in the other columns
tmdb_5000_mov.dropna(inplace=True)
```

Handling duplicates

We are checking for and removing any duplicated rows in the dataset to ensure data accuracy and reliability.

```
# Checking the number of duplicated rows
num_duplicates = tmdb_5000_mov.duplicated().sum()
print(f"Number of duplicated rows: {num_duplicates}")

# Dropping the duplicated rows
data = tmdb_5000_mov.drop_duplicates()

# Verifing the number of rows after dropping duplicates
num_rows = len(data)
print(f"Number of rows after dropping duplicates: {num_rows}")
```

```
Number of duplicated rows: 0
Number of rows after dropping duplicates: 4799
```

```
# Checking the shape
tmdb_5000_mov.shape
```

```
(4799, 21)
```

We are transposing the data frame. This is easier to examine the data closely and see what further action should be takes to clean the data.

```
# Display the first two rows of the dataset

tmdb_5000_mov[:2].T
```

	0	
budget	237000000	
genres	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	[{"id": 12, "name": "Adventure"}

Popularity IMDbPro uses proprietary algorithms that take into account several measures of popularity for people, titles and companies. The primary measure is who and what people are looking at on IMDb. The rankings are updated on a weekly basis, typically by the end of Monday.

This line filters the dataset to exclude rows where 'revenue' = 0. Those rows represent movies that did not generate any revenue, and therefore do not provide relevant information for our analysis.

popularity 150.437577

Let's count the number of movies with a budget equal to 0 and the number of movies with revenue equal to 0.

```
# Counting the number of rows with revenue equal to 0
zero_revenue_count = len(tmdb_5000_mov[tmdb_5000_mov['revenue'] == 0])

# Printing the counts
print("Number of movies with revenue = 0:", zero_revenue_count)
```

```
Number of movies with revenue = 0: 1423
```

title Avatar Pirates of the Caribbean: A

```
# Filtering data where revenue is not 0
tmdb_5000_mov = tmdb_5000_mov[tmdb_5000_mov['revenue']!=0]
```

vote_count 11800

```
# Checking shape
tmdb_5000_mov.shape
```

```
(3376, 21)
```

Preparing the data for modeling

- Split the data into dependent and independent variables.
- Performed a train-test split on the data.
- Scale and encoding the data separately using a pipeline to avoid leakage.

```
# List the variables
list(tmdb_5000_mov)
```

```
['budget',
 'genres',
 'id',
 'keywords',
 'original_language',
 'original_title',
 'overview',
```

```
      'popularity',
      'production_companies',
      'production_countries',
      'release_date',
      'revenue',
      'runtime',
      'spoken_languages',
      'status',
      'title',
      'vote_average',
      'vote_count',
      'tittle',
      'cast',
      'crew']
```

```
# Create a copy for the data
df = tmdb_5000_mov.copy()
```

```
# Selecting only the important variables that are relevant for our analysis.
imp_cols = ['budget', 'genres','popularity','original_language',
            'runtime','vote_average','vote_count','release_date']
```

```
# Creating a dataframe with all the important columns
df = df[imp_cols]
```

```
# Vewing dataframe
df.head()
```

	budget	genres	popularity	original_language	runtime
0	237000000	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	150.437577	en	162.0
1	300000000	[{"id": 12, "name": "Adventure"}, {"id": 14, "...	139.082615	en	169.0

We want to inclide the release month on our model, we will extract the month from the 'release_date' column.

```
# Converting release_date to datetime
df['release_date'] = pd.to_datetime(df['release_date'],
                                              errors='coerce')

# Extracting the month from the release_date column
df['release_date_month'] = df['release_date'].dt.month
```

```
# Dropping release date since we wont need it any more
df.drop('release_date',axis=1,inplace=True)
```

```
# Dropping null values
df.dropna(inplace=True)
```

```
#  Changing the data type to integer
df['release_date_month'] = df['release_date_month'].astype('int')
```

```
#  Changing the data type to object
df['release_date_month'] = df['release_date_month'].astype('object')
```

Handling Genres Column

```
# View numerical columns
```

- The 'genres' column is stored as dictionaries.
- We use the a lambda function to convert the string into actual dictionaries lists using the eval().
- This allows us to access the genre names within the dictionaries. We then drop any rows with null values using dropna().
- Finally, we build a function called get_val() to extract the genre names from the dictionaries.

```
# Extracting genre from the dictionaries
df['genres'] = df['genres'].apply(lambda x: eval(x))
```

```
# Checking the data
df.head()
```

	budget	genres	popularity	original_language	runtime	vo
0	237000000	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	150.437577	en	162.0	
1	300000000	[{'id': 12, 'name': 'Adventure'}, {'id': 14, '...	139.082615	en	169.0	
2	245000000	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	107.376788	en	148.0	
3	250000000	[{'id': 28, 'name': 'Action'}, {'id': 80, 'nam...	112.312950	en	165.0	
4	260000000	[{'id': 28, 'name': 'Action'}, {'id': 12, 'nam...	43.926995	en	132.0	

```
# Dropping null values
df.dropna(inplace=True)
```

```
# Running the function
def get_val(dictionary_list):
    val = [d['name'] for d in dictionary_list]
    return val

# Running a lmabda function for the genres column
from tqdm.notebook import tqdm
tqdm.pandas()

# Apply the get_val function to extract the genre names

df['genres'] = df['genres'].progress_apply(get_val)
```

100% 3376/3376 [00:00<00:00, 61884.47it/s]

```
# Viewing df
df.head()
```

	budget	genres	popularity	original_language	runtime
0	237000000	[Action, Adventure, Fantasy, Science Fiction]	150.437577	en	162.0
1	300000000	[Adventure, Fantasy, Action]	139.082615	en	169.0
2	245000000	[Action, Adventure, Crime]	107.376788	en	148.0
3	250000000	[Action, Crime, Drama, Thriller]	112.312950	en	165.0
4	260000000	[Action, Adventure, Science Fiction]	43.926995	en	132.0

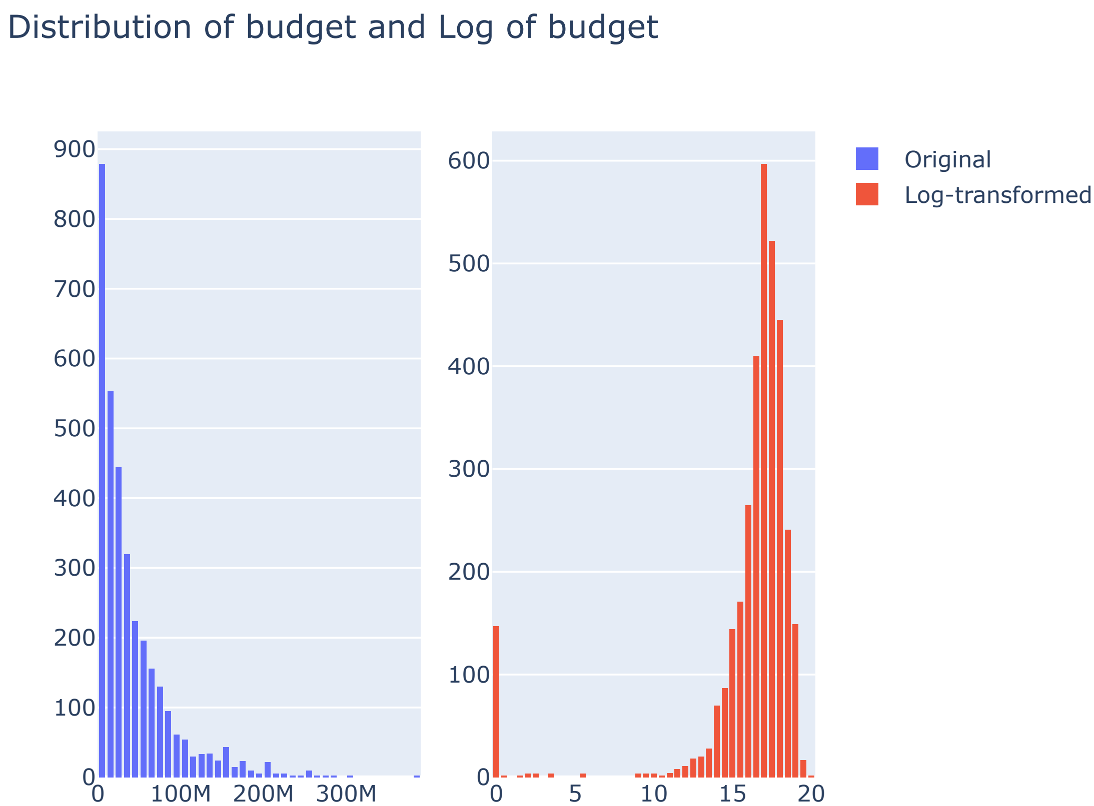
Log Transformation

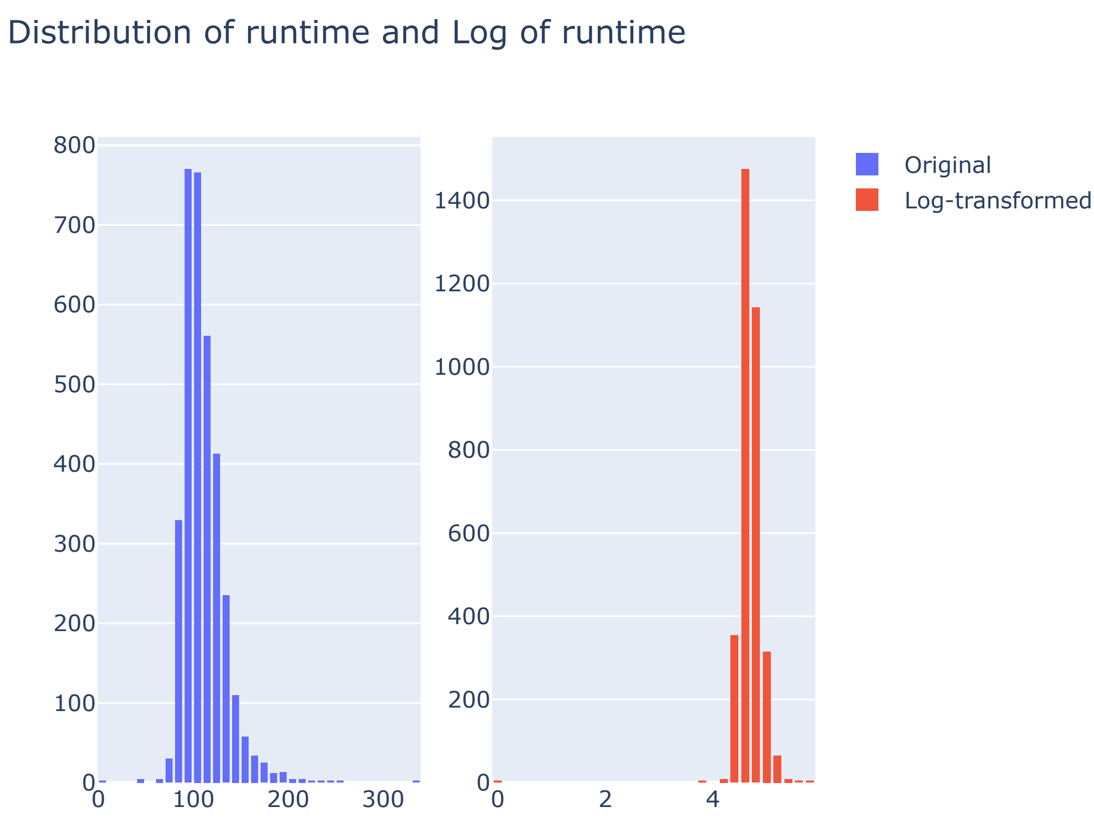
A few variables are skewed in our data. In our case, 'budget' and 'vote_count' as seen below are note normally distributed. Regression model assumes that data is normally distributed.

```
 # Defining the function
def plot_histogram(df, column_name):
  # Create a subplot with 1 row and 2 columns
    fig = make_subplots(rows=1, cols=2)

    # Adding a histogram - first subplot
    fig.add_trace(
        go.Histogram(x=df[column_name], nbinsx=50, name='Original'),
        row=1, col=1  # This trace will go in the first subplot
    )

    # Adding a histogram trace to the second subplot -log-transformed data
    fig.add_trace(
        go.Histogram(x=np.log1p(df[column_name]), nbinsx=50, name='Log-transformed'),
        row=1, col=2  # This trace will go in the second subplot
    )

    # Updating the layout
    fig.update_layout(
```

```
        title_text='Distribution of '+column_name+' and Log of '+column_name,
        bargap=0.2,  # This is the gap between bars
        bargroupgap=0.1
    )
    fig.show()  #
```

```
# Running them all together to compare

plot_histogram(df, 'budget')
plot_histogram(df, 'runtime')
plot_histogram(df, 'vote_average')
plot_histogram(df, 'vote_count')
```


Distribution of budget and Log of budget
Original
Log-transformed
900
800
700
600
500
400
300
200
100
0
0
100M
200M
300M
600
500
400
300
200
100
0
0
5
10
15
20


Distribution of runtime and Log of runtime
Original
Log-transformed
800
700
600
500
400
300
200
100
0
0
100
200
300
1400
1200
1000
800
600
400
200
0
0
2
4

Logarithm transformation makes budget and vote count distribution evenly distributed. Therefore we will use log transformation. It reduces the impact of those really big numbers to balance out our data.

Splitting Variables

```
# Splitting the data into independent and dependent Variables
X = df.drop('popularity',axis=1)
y = df['popularity']
```

```
# Splitting the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.20, random_state=42)
```

```
# Viewing Training data
X_train
```

	budget	genres	original_language	runtime	vote_avera
3206	0	[Drama]	fr	127.0	7
826	55000000	[Drama, Thriller, Crime, Mystery, Romance]	en	123.0	5
3966	2500000	[Action, Crime, Drama, Thriller]	en	92.0	7
1317	38000000	[Action, Drama]	en	129.0	6
2854	0	[Comedy, Romance]	en	93.0	5
...	...	...	...	...	

We need to use PassThroughTransformer in order to transform the 'genres' column into multiple binary columns representing each unuique genre.

It iterates over the unique genres, creating a new binary column for each genre, and populates it with 1 if the movie belongs to that genre, and 0 otherwise.

```
# Storing genres
passthrough_features = ['genres']

# Creating a custom transformer class for passthrough features

class PassthroughTransformer(BaseEstimator):
    def fit(self, X, y = None):
        self.cols = X.columns
        return self

    def transform(self, X, y = None):

        #Creating a copy
        X_ = X.copy()

        # Getting all unique genres from the 'genres' column
        self.all_genre = set(sum(df['genres'],[]))


        # Iterating over each genre and create a new binary column for it
        for gen in tqdm(self.all_genre):
            X_[gen] = X_['genres'].apply(lambda x: 1 if gen in x else 0)

        # Dropping the original 'genres' column
        X_.drop('genres',axis=1,inplace=True)
        return X_

    def get_feature_names(self):
        return list(self.all_genre)

# enc = ColumnTransformer([('pass' , PassthroughTransformer(),
# passthrough_features)])

# https://www.appsloveworld.com/scikit-learn/7/adding-get-feature-names-
```

```
# to-columntransformer-
# https://gist.github.com/tdpetrou/6a97304dd4452a53be98e4f4e93196e6
```

```
set(sum(df['genres'],[]))
```

```
{'Action',
 'Adventure',
 'Animation',
 'Comedy',
 'Crime',
 'Documentary',
 'Drama',
 'Family',
 'Fantasy',
 'Foreign',
 'History',
 'Horror',
 'Music',
 'Mystery',
 'Romance',
 'Science Fiction',
 'Thriller',
 'War',
 'Western'}
```

Confirming Column Type

```
# Remind us of the columns we are deaing with

def get_column_types(data):

   # Initializing empty lists for categorical and numerical
    categorical_cols = []
    numerical_cols = []
# Iterating over each column
    for col in data.columns:
      # Check the data type of the column
        if data[col].dtype == 'object':
          # If it's an object type assign it to categorical column
            categorical_cols.append(col)
        else:
          # Else it is a numerical column
            numerical_cols.append(col)

    return categorical_cols, numerical_cols
```

```
# Passing data to get column type
categorical_cols, numerical_cols = get_column_types(df)

print("Categorical columns:")
print(categorical_cols)
print()
```

```
print("Numerical columns:")
print(numerical_cols)
```

```
Categorical columns:
['genres', 'original_language', 'release_date_month']

Numerical columns:
['budget', 'popularity', 'runtime', 'vote_average', 'vote_count']
```

Separating numerical and categorical

We will be working with two types of data: numerical and categorical. For the pipeline - we need to separate these data types for conducting scaling. Numerical data will be scaled or normalized, while categorical data will undergo encoding techniques.

```
# Storing numerical columns
numeric_cols = ['runtime','vote_average']
```

Pipeline

We use a pipeline to ensure all preprocessing steps are applied only on the training data and prevent any data leakage to the testing (unseeen data).

'Log_transform' evens out data, 'OneHotEncoder' changes categories into numbers, and 'StandardScaler' makes sure all numbers are on the same scale- normalized. We handled Gerne with 'PassthroughTransformer' and therefore it lets it pass through as is.

```
from sklearn.preprocessing import FunctionTransformer
def log_transform(x):
    print(x)
    return np.log1p(x)
```

```
# Create an instance of the OneHotEncoder

transformer = FunctionTransformer(log_transform)
ohe = OneHotEncoder(handle_unknown='ignore')

# Storing one hot encoder in a pipeline
categorical_processing = Pipeline(steps=[('ohe', ohe)])
scaling_processing = Pipeline(steps=[('scale', StandardScaler())])
log_processing = Pipeline(steps=[('transformer', transformer)])
#vote_count_processing = Pipeline(steps=[('transformer', transformer)])

# we need to bring together the preprocessing steps for both cat. and num.
# We create a ColumnTransformer object for preprocessing steps.
# We apply the categorical_processing step to the 'original_language' and
# 'release_date_month' columns,
# and the scaling_processing step to the numeric columns.
```

```
preprocessing = ColumnTransformer(transformers=[
                ('categorical', categorical_processing, ['original_language','release_date_m
        ('log',log_processing,['budget','vote_count']),
        ('numeric', scaling_processing, numeric_cols),
        ('pass' , PassthroughTransformer(), passthrough_features)
                                                ])
# https://stats.stackexchange.com/questions/402470/how-can-i-use-scaling-and-
# log-transforming-together
```

```
# Checking shape for the training data
X_train.shape
```

```
(2700, 7)
```

```
# Checking shape for the testing data
X_test.shape
```

```
(676, 7)
```

By applying fit_transform on the training data and transform on the test data, we make sure that the data is processes for both and we avoid any leakage of information from the test data into the training process.

```
# # Scale and hot encode the train and test data
X_train = preprocessing.fit_transform(X_train)

X_test = preprocessing.transform(X_test)
```

```
          budget  vote_count
3206           0         285
826     55000000         210
3966     2500000          95
1317    38000000          86
2854           0           8
```

```
# pip show scikit-learn
```

```
1400    35000000         458
```

```
# pip install -U scikit-learn
```

We retrieve the feature names for the passthrough transformer and one-hot encoder transformer used in the pipeline. We also define the names of passthrough features and other columns/features.

```
# Retrieving the feature names for the passthrough transformer
genre_label = preprocessing.named_transformers_['pass'].get_feature_names()

# Retrieving the feature names for the one-hot encoder transformer
enc_cat_features = (preprocessing.named_transformers_['categorical']
          ['ohe'].get_feature_names_out()
                              )
# Defining the names of passthrough features
# passthrough_features = ['passthrough1', 'passthrough2', 'passthrough3','passthrough4','pas

# Define the names of other features

other_col = ['runtime','vote_average','vote_count','budget']

# Concatenating all the feature names together

labels = np.concatenate([genre_label, enc_cat_features,other_col])


# https://www.youtube.com/watch?v=NxLfpcfGzns
```

```
X_train = pd.DataFrame(X_train,columns=labels)
X_test = pd.DataFrame(X_test,columns=labels)
```

```
X_train.info()
```

```
 3   Adventure              2700 non-null   float64
 4   Romance                2700 non-null   float64
 5   Crime                  2700 non-null   float64
 6   Western                2700 non-null   float64
```

```
 15  Action                 2700 non-null   float64
 16  Drama                  2700 non-null   float64
 17  Music                  2700 non-null   float64
 18  Family                 2700 non-null   float64
 19  original_language_cn   2700 non-null   float64
 20  original_language_da   2700 non-null   float64
 21  original_language_de   2700 non-null   float64
 22  original_language_el   2700 non-null   float64
 23  original_language_en   2700 non-null   float64
 24  original_language_es   2700 non-null   float64
 25  original_language_fa   2700 non-null   float64
 26  original_language_fr   2700 non-null   float64
 27  original_language_he   2700 non-null   float64
 28  original_language_hi   2700 non-null   float64
 29  original_language_id   2700 non-null   float64
 30  original_language_is   2700 non-null   float64
 31  original_language_it   2700 non-null   float64
 32  original_language_ja   2700 non-null   float64
 33  original_language_ko   2700 non-null   float64
 34  original_language_nb   2700 non-null   float64
 35  original_language_nl   2700 non-null   float64
 36  original_language_no   2700 non-null   float64
 37  original_language_ro   2700 non-null   float64
 38  original_language_ru   2700 non-null   float64
 39  original_language_te   2700 non-null   float64
 40  original_language_th   2700 non-null   float64
 41  original_language_vi   2700 non-null   float64
 42  original_language_zh   2700 non-null   float64
 43  release_date_month_1   2700 non-null   float64
 44  release_date_month_2   2700 non-null   float64
 45  release_date_month_3   2700 non-null   float64
 46  release_date_month_4   2700 non-null   float64
 47  release_date_month_5   2700 non-null   float64
 48  release_date_month_6   2700 non-null   float64
 49  release_date_month_7   2700 non-null   float64
 50  release_date_month_8   2700 non-null   float64
 51  release_date_month_9   2700 non-null   float64
 52  release_date_month_10  2700 non-null   float64
 53  release_date_month_11  2700 non-null   float64
 54  release_date_month_12  2700 non-null   float64
 55  runtime                2700 non-null   float64
 56  vote_average           2700 non-null   float64
 57  vote_count             2700 non-null   float64
 58  budget                 2700 non-null   float64
dtypes: float64(59)
memory usage: 1.2 MB
```

```
# X_train = X_train.toarray()
# X_test = X_test.toarray()
```

Steps for Regression Modeling Approach:

- We will start with a baseline model using ordinary least squares (OLS) regression
- We will select the best features based on their significance
- Then we will implement three machine learning models: Random Forest, XGBoost, and AdaBoost
- We will follow by training a general MLP (Multi-Layer Perceptron) model

- Ultimately, we will evaluate the performance of each model and compare their results

Regression

```
# Checking shape
print('Test data shape:', X_test.shape,'Train data shape:', X_train.shape)
```

```
Test data shape: (676, 59) Train data shape: (2700, 59)
```

```
# Resetting index
y_train.reset_index(drop=True,inplace=True)
```

```
# Running the OLS regression model.
X_train # Using the best features for the model
X_train_int = sm.add_constant(X_train)  # Adding a constant
model_3 = sm.OLS(y_train, X_train).fit() # Fitting the training data
model_3.summary()
```

OLS Regression Results

Dep. Variable:	popularity	**R-squared:**	0.409
Model:	OLS	**Adj. R-squared:**	0.396
Method:	Least Squares	**F-statistic:**	32.05
Date:	Thu, 25 May 2023	**Prob (F-statistic):**	9.53e-256
Time:	20:55:46	**Log-Likelihood:**	-12738.
No. Observations:	2700	**AIC:**	2.559e+04
Df Residuals:	2642	**BIC:**	2.594e+04
Df Model:	57		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Documentary	-18.3392	10.612	-1.728	0.084	-39.148	2.469
War	-22.3593	15.729	-1.421	0.155	-53.203	8.484
Science Fiction	-19.6077	9.414	-2.083	0.037	-38.067	-1.148
Adventure	-14.7225	26.901	-0.547	0.584	-67.473	38.028
Romance	-15.0111	3.154	-4.759	0.000	-21.196	-8.826
Crime	-16.9715	8.555	-1.984	0.047	-33.747	-0.196
Western	-13.9313	26.951	-0.517	0.605	-66.779	38.916
Animation	-21.0662	6.582	-3.201	0.001	-33.972	-8.160
Comedy	-39.0520	27.129	-1.440	0.150	-92.248	14.144
Fantasy	-14.4564	10.525	-1.374	0.170	-35.094	6.181
Mystery	-22.9305	19.239	-1.192	0.233	-60.655	14.794
Foreign	-0.9581	26.868	-0.036	0.972	-53.642	51.726
History	-12.5466	11.492	-1.092	0.275	-35.082	9.989
Horror	-25.3955	8.748	-2.903	0.004	-42.550	-8.241
Thriller	-19.5654	11.323	-1.728	0.084	-41.769	2.638
Action	-34.8691	26.930	-1.295	0.196	-87.676	17.938
Drama	-18.8485	19.280	-0.978	0.328	-56.654	18.957
Music	-21.3554	26.927	-0.793	0.428	-74.156	31.445
Family	-22.7285	26.826	-0.847	0.397	-75.330	29.873
original_language_cn	-12.6154	10.500	-1.202	0.230	-33.204	7.973
original_language_da	-22.8977	27.048	-0.847	0.397	-75.935	30.140
original_language_de	-20.0499	19.130	-1.048	0.295	-57.562	17.462
original_language_el	28.0368	26.887	1.043	0.297	-24.685	80.758
original_language_en	-15.0778	8.371	-1.801	0.072	-31.492	1.337
original_language_es	-34.8356	3.961	-8.794	0.000	-42.603	-27.068
original_language_fa	-36.1535	3.870	-9.342	0.000	-43.742	-28.565
original_language_fr	-35.9044	3.882	-9.248	0.000	-43.517	-28.291
original_language_he	-35.6469	3.875	-9.199	0.000	-43.246	-28.048
original_language_hi	-33.2474	3.879	-8.571	0.000	-40.854	-25.641
original_language_id	-31.8029	3.819	-8.326	0.000	-39.292	-24.313
original_language_is	-34.4208	3.789	-9.085	0.000	-41.850	-26.991
original_language_it	-34.8583	3.657	-9.532	0.000	-42.029	-27.688
original_language_ja	-36.6227	3.670	-9.979	0.000	-43.819	-29.427
original_language_ko	-33.6351	3.755	-8.958	0.000	-40.998	-26.273
original_language_nb	-32.0860	3.908	-8.211	0.000	-39.749	-24.423
original_language_nl	-38.1053	3.781	-10.077	0.000	-45.520	-30.691
original_language_no	-0.1204	0.157	-0.766	0.444	-0.429	0.188
original_language_ro	13.3240	0.486	27.436	0.000	12.372	14.276

original_language_ru 2.8092 0.695 4.044 0.000 1.447 4.171

Selecting best features

original_language_vi -0.3345 3.145 -0.106 0.915 -6.500 5.831

The SelectKBest model evaluates the statistical significance of each feature's relationship with the target variable using F-value. Based on the scores, SelectKBest selects the top k features with the highest scores, which are the features that are most relevant for predicting the target variable.

release_date_month_4 -2.7340 4.209 -0.650 0.516 -10.987 5.519

```
# Selecting the best features for training using SelectKBest
# Selecting the top 5 features
selector = SelectKBest(score_func=f_regression, k=7) # Selecing using select k best
ch = selector.fit(X_train, y_train) # Fitting
X_train_selectk = ch.transform(X_train) # Transforming
```

release_date_month_10 -5.0693 2.796 -1.813 0.070 -10.553 0.414

```
 # fit and transform using selector
X_train_selected = selector.fit_transform(X_train, y_train)

# Get the selected feature indices
selected_features = selector.get_support(indices=True)
# Print the selected feature names
col_sel = X_train.columns[selected_features]
col_sel = list(col_sel)
col_sel
```

```
['original_language_no',
 'original_language_ro',
 'original_language_ru',
 'original_language_te',
 'original_language_zh',
 'release_date_month_1',
 'runtime']
```

```
X_train_selectk = pd.DataFrame(X_train_selectk,columns=col_sel)
X_train_selectk.head()
```

	original_language_no	original_language_ro	original_language_ru	original_language_
0	0.000000	5.655992	0.792856	1.8175
1	17.822844	5.351858	0.602528	-0.4541
2	14.731802	4.564348	-0.872511	0.9088
3	17.453097	4.465908	0.888019	0.0002
4	0.000000	2.197225	-0.824929	-1.1356

```
# X_train.columns[selected_features]
```

```
# Selecting for test as well
X_test_selectk = selector.transform(X_test)
```

```
# Checking training data
X_train_selectk.shape
```

```
(2700, 7)
```

```
# Checking testing data
X_test_selectk.shape
```

```
(676, 7)
```

```
# See the scaling of X_train

# Resetting index

y_train.reset_index(drop=True,inplace=True)

# Adding a constant column to the selected features

X_train_selectk = X_train_selectk.copy()

X_train_int = sm.add_constant(X_train_selectk)

# Creating the OLS model
model = sm.OLS(y_train, X_train_int)

# Fitting the model
OLS_SF = model.fit()

# Printing the summary
OLS_SF.summary()
```

OLS Regression Results

Dep. Variable:	popularity	**R-squared:**	0.396
Model:	OLS	**Adj. R-squared:**	0.395
Method:	Least Squares	**F-statistic:**	252.4
Date:	Thu, 25 May 2023	**Prob (F-statistic):**	2.04e-289
Time:	20:55:46	**Log-Likelihood:**	-12767.
No. Observations:	2700	**AIC:**	2.555e+04
Df Residuals:	2692	**BIC:**	2.560e+04
Df Model:	7		
Covariance Type:	nonrobust		

Interpretation of SlectK OLS

- The model explains approximately 54.3% of the variability in the popularity.

original_language_no and release_date_month_1, have significant effects on popularity (low p-values).

Evaluating Results

We build a function to calculate and return the mean squared error (MSE) and root mean squared error (RMSE) score. It also adds the results to a pandas dataframe for later use. includes a conditional if the model is from statsmodels - it addes a constant.

Notes:

```
# Creating an empty dataframe
results_df_ml = pd.DataFrame([],columns=['Model','MSE','RMSE'])
# Initialize an empty string for the name of the model
mname = ''

# Function
def  get_results(reg,x_test):
    mname = ''
    # If the reg model is from statmodels
    if 'statsmodels' in str(type(reg)):
        x_test = sm.add_constant(x_test)
        star = '*'
        mname = 'OLS - SelectK'

    # Getting predicted values
    y_pred = reg.predict(x_test)
    # Calculate mean squared error
    mse = mean_squared_error(y_test, y_pred)

    # Calculate root mean squared error
    rmse = mean_squared_error(y_test, y_pred,squared=False)

    print('mse',mse)
    print('rmse',rmse)
```

```
    # Obtaining the name for the reg model
    if not mname:
        mname = type(reg).__name__

    # Store the results in the DataFrame
    results_df_ml.loc[len(results_df_ml)] = [mname,mse,rmse]
    return results_df_ml
# https://datascience.stackexchange.com/questions/26555/valueerror-shapes-1-10-
# and-2-not-aligned-10-dim-1-2-dim-0
# https://stackoverflow.com/questions/54003129/valueerror-shapes-993-228-and-1
# -228-not-aligned-228-dim-1-1-dim-0
```

```
# Checking shape
X_test_selectk.shape
```

```
(676, 7)
```

```
# Run the function

# X_test_selectk = X_test.copy()
get_results(OLS_SF, X_test_selectk)
```

```
mse 777.8235242864788
rmse 27.889487702115986
```

	Model	MSE	RMSE
0	OLS - SelectK	777.823524	27.889488

Machine Learning Models

Random Forest Random Forest is an algorithm that combines multiple decision trees to make predictions.

It hands complex datasets. Random Forest aggregates the predictions of multiple trees, and that way it reduces overfitting and improves prediction accuracy.

XGBoost

XGBoost is a gradient boosting algorithm. It is used by training a series of weak learners of decision trees, and adding them to the ensemble. It focuses on the mistakes made by the previous learners, allowing the model to improve its predictions. It employs gradient boosting, where the next models are trained to minimize the errors of the previous models.

AdaBoost

AdaBoost, similar to XGBoost, also learns from weak learners, but it differs in the way it combines their predictions. While XGBoost uses gradient boosting to optimize the overall model, AdaBoost assigns weights to the weak learners based on their performance and focuses on samples with higher error.

Machine learning function

Initializing the models (Random Forest, XGBoost, and AdaBoost)

- Creating an empty DataFrame to store the results.
- Iterating through each model and performs the following:
    1. Fitting the model using the training data.
    2. Obtaining predictions on the test data.
    3. Calculating the (MSE) and (RMSE) between the predicted and actual values.
    4. Adding the model name, MSE, and RMSE to the results DataFrame.
- Returning the results DataFrame containing the model names, MSE, and RMSE for each model.

```
model_dict = {
     'Random Forest': RandomForestRegressor(random_state=42),
     'XGBoost': XGBRegressor(),
     'AdaBoost': AdaBoostRegressor(random_state=42)

 }
```

```
def run_regression_models(model_dict,X_train, y_train, X_test, y_test):
    # Initialize the models

    # Initialize the DataFrame to store results

    # Loop through each model
    for model_name, model in tqdm(model_dict.items()):
        # Fit the model
        model.fit(X_train, y_train)

        # Make predictions
        y_pred = model.predict(X_test)

        # Calculate MSE and RMSE
        mse = mean_squared_error(y_test, y_pred)
        rmse = mean_squared_error(y_test, y_pred, squared=False)

#         results_df_ml
        # Add results to the DataFrame

        results_df_ml.loc[len(results_df_ml)] = [model_name,mse,rmse]
#         results_df_ml = results_df_ml.append({
#             'Model': model_name,
#             'MSE': mse,
#             'RMSE': rmse
#         }, ignore_index=True)

    return results_df_ml
```

```
# Viewing resutls of the ML models
results_df_ml = run_regression_models(model_dict,X_train, y_train, X_test, y_test)
```

```
results_df_ml
```

100% 3/3 [00:19<00:00, 6.05s/it]

	Model	MSE	RMSE
0	OLS - SelectK	777.823524	27.889488
1	Random Forest	438.556625	20.941744
2	XGBoost	628.882154	25.077523
3	AdaBoost	763.496747	27.631445

OLS Select performed better than the othermodels in predicting movie popularity. We will explore the potential of a General Multilayer Perceptron (MLP) model, a deep learning approach, to further improve our predictions.

Multilayer Perceptron MLP

MLP is a type of neural network that consists of 3 or more layers of neurons. MLP, short for MLP, is a type of neural network that is composed of multiple layers of nodes, with each node being a simple computational unit that performs a mathematical operation. The MLP takes input data, processes it through the layers of nodes, and produces output predictions.

During training, the weights between nodes are adjusted through a process called backpropagation. The weights are updated to minimize the difference between the predicted outputs and the actual outputs.

```
# Setting the input shape
input_shape = (X_train.shape[1],)
print(f'Feature shape: {input_shape}')
```

```
Feature shape: (59,)
```

We utilized two callbacks:

1. ReduceLROnPlateau adjusts the learning rate based on the validation loss.
2. EarlyStopping stops training if the mean squared error improvement is below a certain threshold.

```
# Initializing callback
callbacks = [ ReduceLROnPlateau(monitor='val_loss', patience=5, cooldown=0),
              EarlyStopping(monitor='mean_squared_error',
                            min_delta=1e-4,
                            patience=5)
                            ]
```

```
# Create the model
mlp_model = Sequential()

# Adding the input layer with 16 neurons and ReLU activation
mlp_model.add(Dense(16, input_shape=input_shape, activation='relu'))

# Adding a hidden layer with 8 neurons and ReLU activation
mlp_model.add(Dense(8, activation='relu'))

# Adding the output layer with 1 neuron and linear activation
mlp_model.add(Dense(1, activation='linear'))

# Configure the model and start training
mlp_model.compile(loss='mean_squared_error', optimizer='adam',
                  metrics=['mean_squared_error'])

# Train the model on the training
mlp_model.fit(X_train, y_train, epochs=5, batch_size=32,
          verbose=1, validation_split=0.2,callbacks=callbacks)
```

```
Epoch 1/5
68/68 [==============================] - 4s 13ms/step - loss: 2215.5647 - mean_squared_
Epoch 2/5
68/68 [==============================] - 1s 8ms/step - loss: 1969.4706 - mean_squared_e
Epoch 3/5
68/68 [==============================] - 0s 5ms/step - loss: 1680.6898 - mean_squared_e
Epoch 4/5
68/68 [==============================] - 0s 4ms/step - loss: 1399.7075 - mean_squared_e
Epoch 5/5
68/68 [==============================] - 1s 7ms/step - loss: 1250.7611 - mean_squared_e
<keras.callbacks.History at 0x7ff7b781d090>
```

```
# Making predictions on the test set
y_pred_mlp = mlp_model.predict(X_test)
```

```
22/22 [==============================] - 0s 6ms/step
```

```
# Calculating mean squared error
mlp_mse = mean_squared_error(y_test, y_pred_mlp)
print('Mean Squared Error:', mlp_mse)
```

```
Mean Squared Error: 1186.3735101028667
```

```
# Seeing the shape
X_train.shape
```

```
(2700, 59)
```

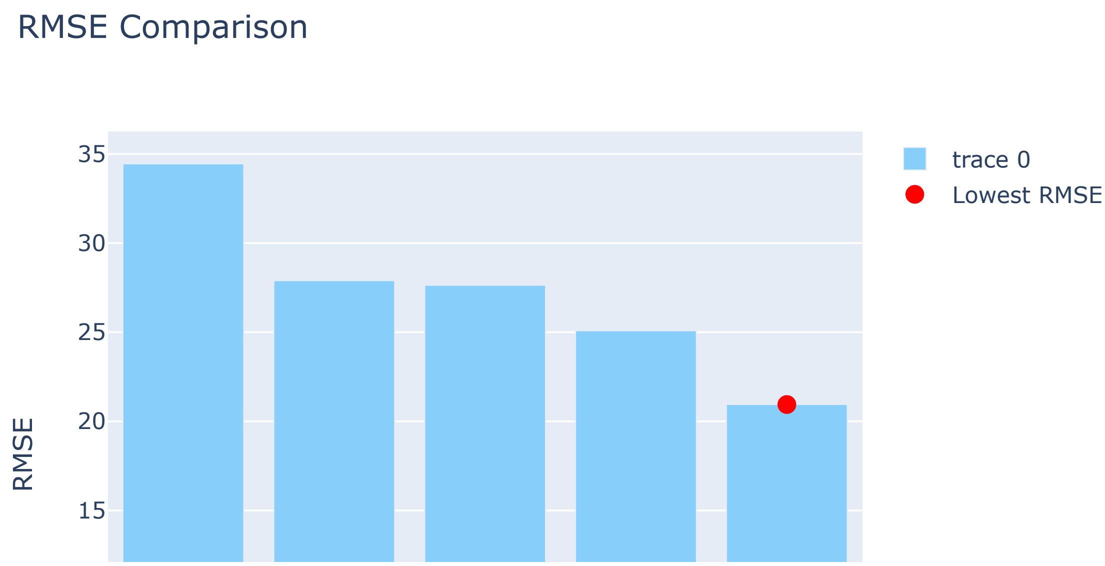
```
# Calculating root mean squared error
mlp_rmse = mean_squared_error(y_test, y_pred_mlp,squared=False)
print('Root Mean Squared Error:', round(mlp_rmse,3))
```

```
Root Mean Squared Error: 34.444
```

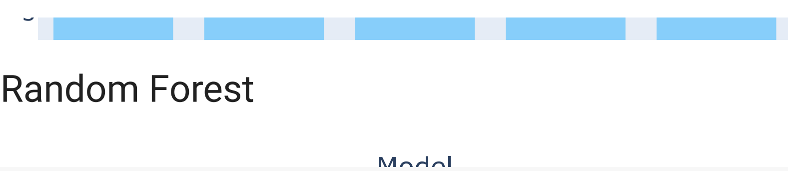
```
# Adding results for mlp model
mlp_results = {'Model': 'MLP', 'MSE': mlp_mse, 'RMSE': mlp_rmse}
results_df_ml = results_df_ml.append(mlp_results, ignore_index=True)
results_df_ml
```

	Model	MSE	RMSE
0	OLS - SelectK	777.823524	27.889488
1	Random Forest	438.556625	20.941744
2	XGBoost	628.882154	25.077523
3	AdaBoost	763.496747	27.631445
4	MLP	1186.373510	34.443773

```
# Sorting by RMSE in descending order
results_df_ml_sorted = results_df_ml.sort_values(by='RMSE',
                                                 ascending=False)

# Creating a bar plot with RMSE values in descending order
fig = go.Figure(data=go.Bar(x=results_df_ml_sorted['Model'],
                            y=results_df_ml_sorted['RMSE'],
                            marker_color='lightskyblue'))

# Find the index of the lowest RMSE value
lowest_rmse_index = results_df_ml_sorted['RMSE'].idxmin()

# Add a marker for the lowest RMSE value
fig.add_trace(go.Scatter(x=[results_df_ml_sorted['Model'][lowest_rmse_index]],
                         y=[results_df_ml_sorted['RMSE'][lowest_rmse_index]],
                         mode='markers', marker=dict(color='red', size=10),
                         name='Lowest RMSE'))

# Customize the layout
fig.update_layout(title='RMSE Comparison',
                  xaxis_title='Model',
                  yaxis_title='RMSE',
                  showlegend=True)


fig.show()
```



Random Forest was the top performer. While MLP did not perform as well, we will explore hyperparameter tuning for MLP to see if we can enhance its performance.



Hypertune Random Forest

```
# Seeing result
results_df_ml
```

	Model	MSE	RMSE
0	OLS - SelectK	777.823524	27.889488
1	Random Forest	438.556625	20.941744
2	XGBoost	628.882154	25.077523
3	AdaBoost	763.496747	27.631445
4	MLP	1186.373510	34.443773

We will explore different combinations of these parameters, we aim to find the optimal cmbinations that can improve the model's performance.

```
# https://www.kaggle.com/code/sociopath00/random-forest-using-gridsearchcv
# Setting up parameters
parameters = {
    'n_estimators': [100, 150, 200, 250, 300,400],
    'max_depth': [1,2,3,4,5,7,9],
}
```

```
# Creating the Random Forest regressor
rf = RandomForestRegressor(n_jobs=-1) #
```

We will use 5-fold cross-validation to evaluate the performance of each parameter combination.

```
# Performing randomized search with progress bar
gird_search_rf = GridSearchCV(estimator=rf,
                                      param_grid=parameters,
                                      verbose=1,n_jobs=-1,
                                      cv=5)
```

```
# Fitting grid search
rf_model = gird_search_rf.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 42 candidates, totalling 210 fits
```

```
# Getting best estimator
grid_rf = rf_model.best_estimator_
grid_rf
```

```
▾ RandomForestRegressor
RandomForestRegressor(max_depth=2, n_estimators=150, n_jobs=-1)
```

```
# Fitting grid
grid_rf.fit(X_train, y_train)
```

```
▾ RandomForestRegressor
RandomForestRegressor(max_depth=2, n_estimators=150, n_jobs=-1)
```

```
# Predicting
y_pred_rf_tuned = grid_rf.predict(X_test)
```

```
# Get the best hyperparameters and model
# best_params_rf = random_search_rf.best_params_
# best_model_rf = random_search_rf.best_estimator_
```

```
# Calculating RMSE
tuned_rf_rmse = mean_squared_error(y_test, y_pred_rf_tuned,squared=False)
print('Root Mean Squared Error:', round(tuned_rf_rmse,3))
```

```
Root Mean Squared Error: 21.493
```

```
# Calculating MSE
tuned_rf_mse = mean_squared_error(y_test, y_pred_rf_tuned,squared=True)
print('Mean Squared Error:', round(tuned_rf_mse,3))
```

```
Mean Squared Error: 461.934
```

```
# Adding results for MLP model
grid_rf_results = {'Model': 'RF_Tuned', 'MSE': tuned_rf_mse,
                   'RMSE': tuned_rf_rmse }

results_df_ml = results_df_ml.append(grid_rf_results,
                                   ignore_index=True)
# Sorting it out
results_df_ml.sort_values('RMSE', inplace =True)
results_df_ml
```

	Model	MSE	RMSE
1	Random Forest	438.556625	20.941744
5	RF_Tuned	461.934018	21.492650
2	XGBoost	628.882154	25.077523
3	AdaBoost	763.496747	27.631445
0	OLS - SelectK	777.823524	27.889488
4	MLP	1186.373510	34.443773

Hypertune MLP

```
# Create the MLP regressor
mlp = mlp_model
```

```
# Installign scikeras
# pip install scikeras
```

We define a parameter grid of different options for optimizers, epochs, and batch sizes. Then we aplly GridSearchCV with the specified parameter grid to find the best combination of hyperparameters.

```
# Keras model using MLP architecture
Kmodel = KerasRegressor(build_fn=mlp)

# Hyperparameter options for grid search
optimizers = ['rmsprop', 'adam']

epochs = np.array([50, 100])
batches = np.array([10, 20])
param_grid = dict(optimizer=optimizers, epochs=epochs, batch_size=batches)

# Grid search using cross-validation
grid_search_mlp = GridSearchCV(estimator=Kmodel, param_grid=param_grid,
                               n_jobs=-1,cv=3,verbose=2)


#Source: https://www.kaggle.com/code/shujunge/gridsearchcv-with-keras
```

```
      # https://stackoverflow.com/questions/60350049/tensorflow-fit-
      # gives-typeerror-cannot-clone-object-error
```

```
# Returns the valid parameters for the Kmodel
Kmodel.get_params().keys()
```

```
dict_keys(['model', 'build_fn', 'warm_start', 'random_state', 'optimizer', 'loss',
'metrics', 'batch_size', 'validation_batch_size', 'verbose', 'callbacks',
'validation_split', 'shuffle', 'run_eagerly', 'epochs'])
```

```
# Fitting the mdoel
grid_search_mlp.fit(X_train, y_train)
```

```
Epoch 32/100
270/270 [==============================] - 0s 2ms/step - loss: 657.5400 - mean_squared_
Epoch 33/100
270/270 [==============================] - 1s 2ms/step - loss: 658.0704 - mean_squared_
Epoch 34/100
270/270 [==============================] - 0s 2ms/step - loss: 653.1479 - mean_squared_
Epoch 35/100
270/270 [==============================] - 0s 2ms/step - loss: 655.6564 - mean_squared_
Epoch 36/100
270/270 [==============================] - 0s 2ms/step - loss: 646.8091 - mean_squared_
Epoch 37/100
270/270 [==============================] - 0s 2ms/step - loss: 656.8650 - mean_squared_
Epoch 38/100
270/270 [==============================] - 0s 2ms/step - loss: 650.1600 - mean_squared_
Epoch 39/100
270/270 [==============================] - 0s 2ms/step - loss: 644.2171 - mean_squared_
Epoch 40/100
270/270 [==============================] - 1s 2ms/step - loss: 642.0815 - mean_squared_
Epoch 41/100
270/270 [==============================] - 1s 2ms/step - loss: 642.9159 - mean_squared_
Epoch 42/100
270/270 [==============================] - 1s 2ms/step - loss: 644.9333 - mean_squared_
Epoch 43/100
270/270 [==============================] - 1s 2ms/step - loss: 637.8125 - mean_squared_
Epoch 44/100
270/270 [==============================] - 1s 2ms/step - loss: 638.4834 - mean_squared_
Epoch 45/100
270/270 [==============================] - 1s 2ms/step - loss: 635.4930 - mean_squared_
Epoch 46/100
270/270 [==============================] - 0s 2ms/step - loss: 631.2529 - mean_squared_
Epoch 47/100
270/270 [==============================] - 0s 2ms/step - loss: 634.9278 - mean_squared_
Epoch 48/100
270/270 [==============================] - 0s 2ms/step - loss: 635.8853 - mean_squared_
Epoch 49/100
270/270 [==============================] - 0s 2ms/step - loss: 631.2177 - mean_squared_
Epoch 50/100
270/270 [==============================] - 0s 2ms/step - loss: 628.3909 - mean_squared_
Epoch 51/100
270/270 [==============================] - 0s 2ms/step - loss: 629.5685 - mean_squared_
Epoch 52/100
270/270 [==============================] - 0s 2ms/step - loss: 623.4545 - mean_squared_
Epoch 53/100
270/270 [==============================] - 0s 2ms/step - loss: 628.7056 - mean_squared_
Epoch 54/100
270/270 [==============================] - 0s 2ms/step - loss: 620.0629 - mean_squared_
Epoch 55/100
270/270 [==============================] - 0s 2ms/step - loss: 620.2781 - mean_squared_
Epoch 56/100
270/270 [==============================] - 0s 2ms/step - loss: 618.6511 - mean_squared_
Epoch 57/100
270/270 [==============================] - 0s 2ms/step - loss: 623.1468 - mean_squared_
Epoch 58/100
270/270 [==============================] - 0s 2ms/step - loss: 618.5023 - mean_squared_
Epoch 59/100
270/270 [==============================] - 0s 2ms/step - loss: 615.7458 - mean_squared_
Epoch 60/100
270/270 [==============================] - 0s 2ms/step - loss: 612.5548 - mean_squared_
Epoch 61/100
270/270 [==============================] - 1s 2ms/step - loss: 617.0449 - mean_squared_
Epoch 62/100
```

```
270/270 [==============================] - 0s 2ms/step - loss: 615.1420 - mean_squared
```

```
# Get the best hyperparameters and model
best_params_mlp = grid_search_mlp.best_params_
best_model_mlp = grid_search_mlp.best_estimator_
```

```
Epoch 65/100
```

```
# Evaluate the best model
test_loss_mlp = mean_squared_error(y_test, best_model_mlp.predict(X_test))
test_loss_mlp
```

```
68/68 [==============================] - 0s 1ms/step
669.6236308655398
```

```
Epoch 69/100
```

```
# Getting predicted values
y_pred_mlp_tuned = best_model_mlp.predict(X_test)
```

```
68/68 [==============================] - 0s 1ms/step
```

```
Epoch 72/100
```

```
# Calculating RSME
test_loss_mlp = mean_squared_error(y_test, y_pred_mlp_tuned,squared= False)
print('Root Mean Squared Error:', round(test_loss_mlp,3))
```

```
Root Mean Squared Error: 25.877
270/270 [==============================] - 0s 2ms/step - loss: 597.2983 - mean_squared_
```

```
results_all = get_results(best_model_mlp, X_test)
results_all.sort_values('RMSE', inplace=True)
```

```
68/68 [==============================] - 0s 1ms/step
mse 669.6236308655398
rmse 25.877086985701073
270/270 [==============================] - 0s 2ms/step - loss: 584.5554 - mean_squared_
```

```
results_all
```

	Model	MSE	RMSE
1	Random Forest	438.556625	20.941744
5	RF_Tuned	461.934018	21.492650
2	XGBoost	628.882154	25.077523
6	KerasRegressor	669.623631	25.877087
3	AdaBoost	763.496747	27.631445
0	OLS - SelectK	777.823524	27.889488
4	MLP	1186.373510	34.443773

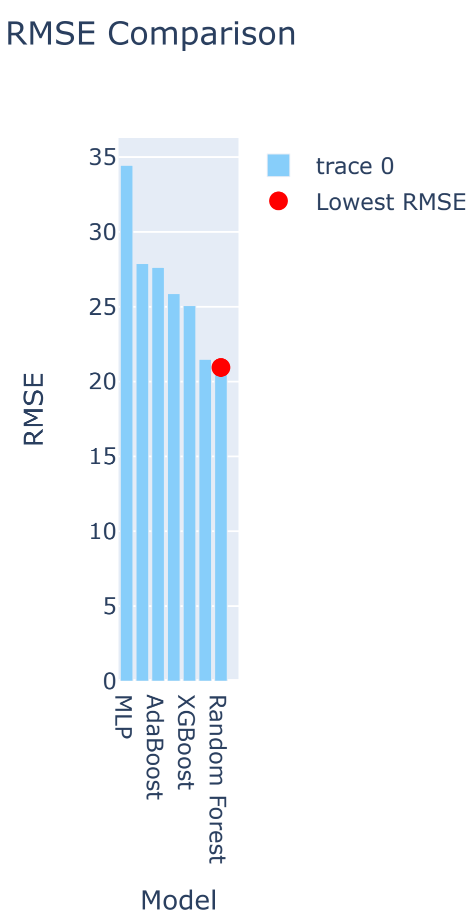
```
Epoch 89/100
```

The plot visulaize the performance of the Random Forest model in predicting movie popularity.

```
270/270 [==============================] - 0s 2ms/step - loss: 581.2440 - mean squared
```

```
import plotly.graph_objects as go

# Sorting by RMSE in descending order
```

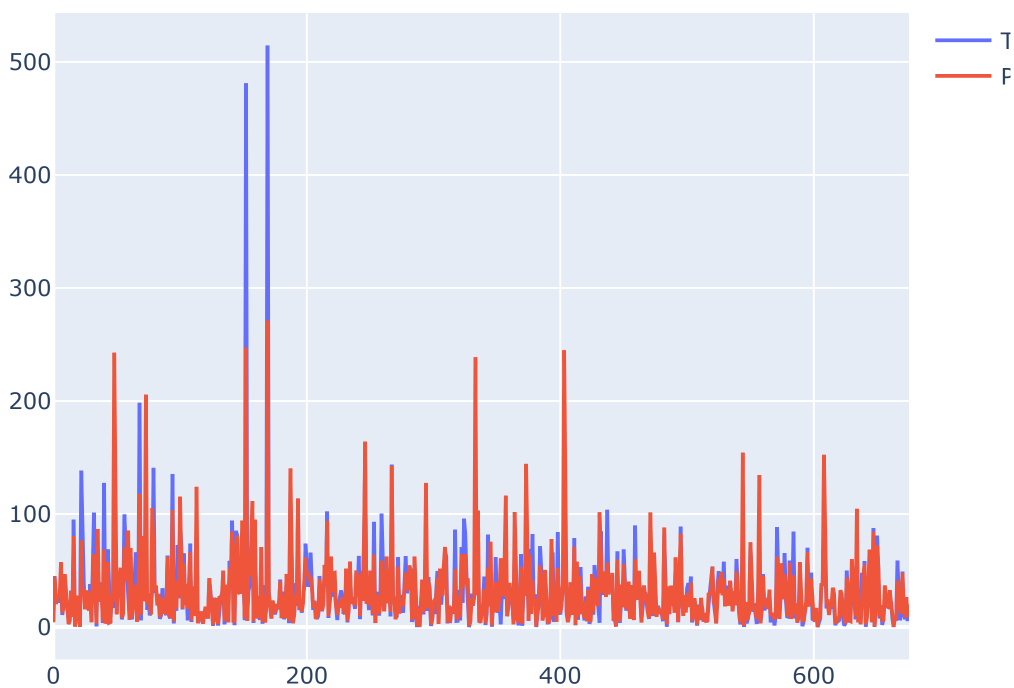
```
# Sorting by RMSE in descending order
results_df_ml_sorted = results_df_ml.sort_values(by='RMSE',
                                                  ascending=False)

# Creating a bar plot with RMSE values in descending order
fig = go.Figure(data=go.Bar(x=results_df_ml_sorted['Model'],
                            y=results_df_ml_sorted['RMSE'],
                            marker_color='lightskyblue'))

# Find the index of the lowest RMSE value
lowest_rmse_index = results_df_ml_sorted['RMSE'].idxmin()

# Add a marker for the lowest RMSE value
fig.add_trace(go.Scatter(x=[results_df_ml_sorted['Model'][lowest_rmse_index]],
                         y=[results_df_ml_sorted['RMSE'][lowest_rmse_index]],
                         mode='markers', marker=dict(color='red', size=10),
                         name='Lowest RMSE'))

# Customize the layout
fig.update_layout(title='RMSE Comparison',
                  xaxis_title='Model',
                  yaxis_title='RMSE',
                  showlegend=True)

# Show the plot
fig.show()
```



```
# Random Forest
reg_rf = RandomForestRegressor(random_state=42)
reg_rf.fit(X_train, y_train)
y_pred_rf = reg_rf.predict(X_test)
mean_squared_error(y_test, y_pred_rf, squared=False)
```

```
20.941743597514034
```

```
fig = go.Figure()
fig.add_trace(go.Scatter(x=list(range(0, len(y_pred_rf))),
                         y=y_test,
                         name='TRUE'))
fig.add_trace(go.Scatter(x=list(range(0, len(y_pred_rf))),
                         y=y_pred_rf,
                         name='PRED'))

fig.show()
```



Features of Importance

Now we will analyze the importance of different features.

```
feature_names = list(X_train)
feature_names
```

```
 'War',
 'Science Fiction',
 'Adventure',
 'Romance',
 'Crime',
 'Western',
 'Animation',
 'Comedy',
 'Fantasy',
 'Mystery',
 'Foreign',
 'History',
 'Horror',
 'Thriller',
 'Action',
 'Drama',
 'Music',
 'Family',
 'original_language_cn',
 'original_language_da',
 'original_language_de',
 'original_language_el',
 'original_language_en',
 'original_language_es',
 'original_language_fa',
 'original_language_fr',
 'original_language_he',
 'original_language_hi',
 'original_language_id',
 'original_language_is',
 'original_language_it',
 'original_language_ja',
 'original_language_ko',
 'original_language_nb',
 'original_language_nl',
 'original_language_no',
 'original_language_ro',
 'original_language_ru',
 'original_language_te',
 'original_language_th',
 'original_language_vi',
 'original_language_zh',
 'release_date_month_1',
 'release_date_month_2',
 'release_date_month_3',
 'release_date_month_4',
 'release_date_month_5',
 'release_date_month_6',
 'release_date_month_7',
 'release_date_month_8',
 'release_date_month_9',
 'release_date_month_10',
 'release_date_month_11',
 'release_date_month_12',
 'runtime',
 'vote_average',
```

```
        'budget']
```

```
# Define Random Forest Tuned model
forest = reg_rf
```

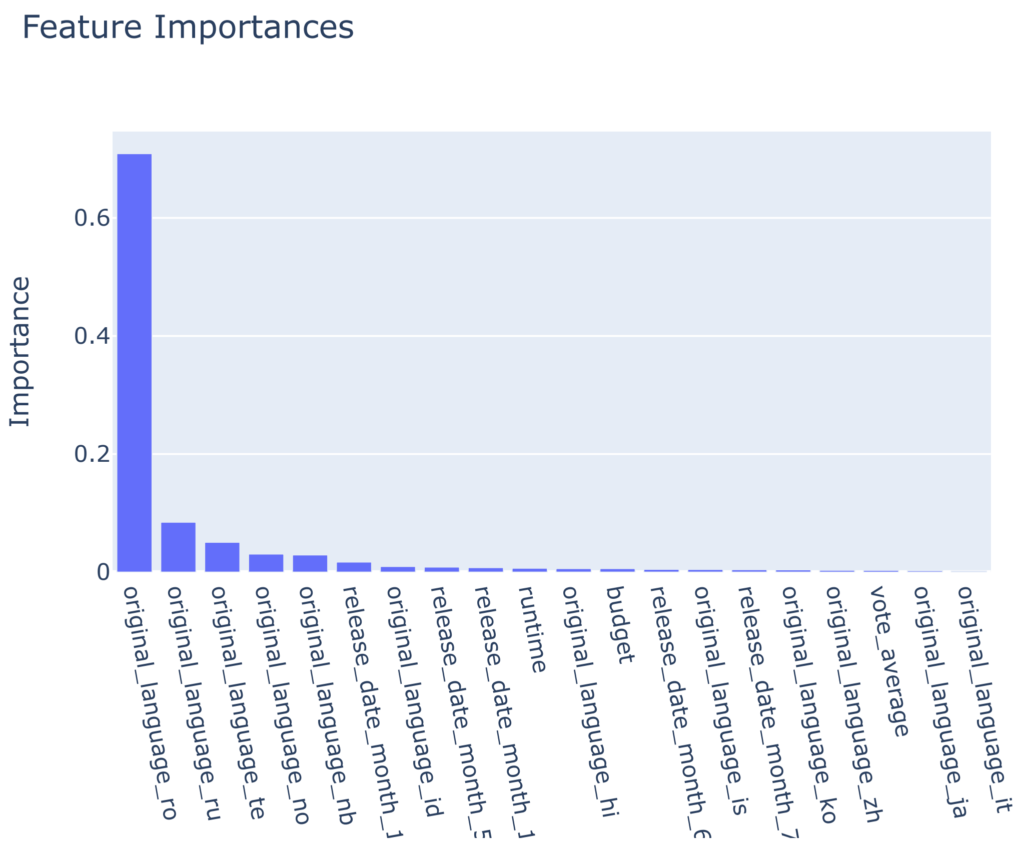
```
# Calculating the feature importances

importances = forest.feature_importances_

std = np.std([tree.feature_importances_ for tree in forest.estimators_], axis=0)
```

```
# Creating a pandas series for feature importances

forest_importances = pd.Series(importances, index=feature_names)
```

```
# Sort the features by top 10 most important
forest_importances = forest_importances.sort_values(ascending=False)[:20]
forest_importances
```

```
original_language_ro     0.708844
original_language_ru     0.084433
original_language_te     0.050409
original_language_no     0.030520
original_language_nb     0.028990
release_date_month_1     0.016946
original_language_id     0.009423
release_date_month_5     0.008462
release_date_month_12    0.007501
runtime                  0.006471
original_language_hi     0.005874
budget                   0.005864
release_date_month_6     0.004557
original_language_is     0.004386
release_date_month_7     0.004016
original_language_ko     0.003744
original_language_zh     0.003004
vote_average             0.002899
original_language_ja     0.002294
original_language_it     0.001655
dtype: float64
```

```
#Plot teh features

fig = go.Figure(data=go.Bar(x=forest_importances.index,
                            y=forest_importances.values))
fig.update_layout(title="Feature Importances", xaxis_title="Features",
                  yaxis_title="Importance")
fig.update_xaxes(tickangle=80)
fig.show()
```



Conclusion

RMSE quantifies the average distance between the predicted values and the actual values, and indicates how well the model's predictions match the true values. A lower RMSE value indicates that the model's predictions are closer to the actual values, suggesting higher accuracy and better performance.

The top-performing model in predicting movie popularity is the regular Random Forest (RF) model followed by the hyperparameter-tuned RF model and the OLS model with SelectKBest features The MLP model did not perform as well as the RF models in predicting movie popularity likely because the data was not large enough.

Save and Load the Model for the Demo

```
from joblib import Parallel, delayed
import joblib

# Save the model as a pickle in a file
joblib.dump(reg_rf,'reg_rf.pkl')

joblib.dump(preprocessing, 'preprocessing.pkl')

# Load the model from the file
rf_from_joblib = joblib.load('reg_rf.pkl')

cleaner_from_joblib = joblib.load('preprocessing.pkl')
```