Predicting Animation Movie Release Date

```
import pandas as pd
```

Exploring the data

```
path = "/content/Animation_Movies.csv"
data = pd.read_csv("Animation_Movies.csv")
#data = pd.read_csv(path)
data
```

1	14160	Up	7.949	18857	Released	2009-05-28	735099
2	12	Finding Nemo	7.824	18061	Released	2003-05-30	940335
3	354912	Coco	8.222	17742	Released	2017-10-27	800526
4	10681	WALL·E	8.078	17446	Released	2008-06-22	521311
...	...	...	...	...	...	...	
51940	656677	Белозубка	0.000	0	Released	2018-12-20	
51941	657149	Shimajiro to Ururu no Heroland	0.000	0	Released	2019-03-15	
51942	656945	Robo Force: The Revenge of Nazgar	0.000	0	Released	1984-12-08	
51943	656893	Beginning Responsibility: A Lunchroom Goes Ban...	0.000	0	Released	1978-01-01	
51944	656966	Natural Selection	0.000	0	Released	2019-08-20	

51945 rows × 23 columns

```
data.isnull().sum()
```

```
id                         0
title                      1
vote_average               0
vote_count                 0
status                     0
release_date            2137
revenue                    0
runtime                    0
adult                      0
backdrop_path          36110
budget                     0
homepage               43692
imdb_id                22393
original_language          0
original_title             1
overview                6079
popularity                 0
poster_path            14011
tagline                47267
genres                     0
production_companies   22547
production_countries   12245
spoken_languages       18127
dtype: int64
```

```
# Filter out observations where revenue is non-positive
data = data[data['revenue'] > 0]

# Check the shape of the DataFrame after filtering
print(data.shape)
```

```
(1100, 23)
```

```
# Calculate null percentages
null_percentages = (data.isnull().sum() / len(data)) * 100

print("Null Percentages:")
print(null_percentages)
```

```
Null Percentages:
id                       0.000000
title                    0.000000
vote_average             0.000000
vote_count               0.000000
status                   0.000000
release_date             1.090909
revenue                  0.000000
runtime                  0.000000
adult                    0.000000
backdrop_path           11.363636
budget                   0.000000
homepage                49.000000
imdb_id                  6.363636
original_language        0.000000
original_title           0.000000
overview                 4.000000
popularity               0.000000
poster_path              3.454545
tagline                 36.727273
genres                   0.000000
production_companies     7.363636
production_countries     4.909091
spoken_languages         3.636364
dtype: float64
```

Remove predictors where more than 25% of the data is missing and isnt relevant to our analysis

```
animation = data.drop(["backdrop_path","homepage","imdb_id", "poster_path", "tagline", "production_companies","production_countries" ],axis=
print(animation.shape)
```

```
(1100, 16)
```

```
# Drop the rows with the remaining null values, imputing doesnt make sense in this context
animation_processed = animation.dropna()
```

```
animation_processed.shape
```

```
(1022, 16)
```

```
# Calculate null percentages
null_percentages = (animation_processed.isnull().sum() / len(animation_processed)) * 100

print("Null Percentages:")
print(null_percentages)
```

```
Null Percentages:
id                  0.0
title               0.0
vote_average        0.0
vote_count          0.0
status              0.0
release_date        0.0
revenue             0.0
runtime             0.0
adult               0.0
budget              0.0
original_language   0.0
original_title      0.0
overview            0.0
popularity          0.0
genres              0.0
spoken_languages    0.0
dtype: float64
```

```
# data dimensions after removing NA values
print(animation_processed.shape)
```

```
(1022, 16)
```

14855 observations and 19 features

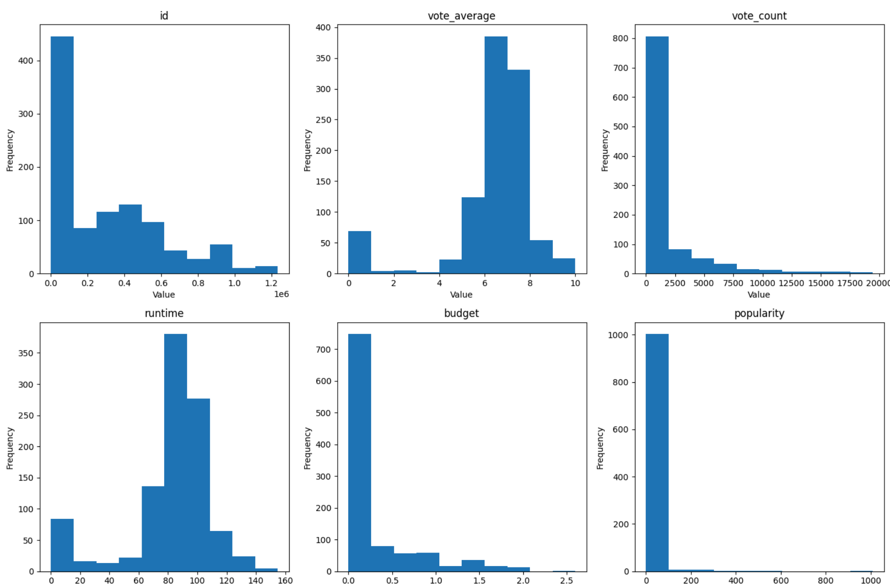
Preprocessing

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Convert release_date column to datetime
animation_processed['release_date'] = pd.to_datetime(animation_processed['release_date'])
# Extract temporal features
animation_processed['release_month'] = animation_processed['release_date'].dt.month
animation_processed['release_year'] = animation_processed['release_date'].dt.year
animation_processed['release_day_of_week'] = animation_processed['release_date'].dt.dayofweek  # Monday=0, Sunday=6
animation_processed['release_season'] = (animation_processed['release_month'] % 12 + 3) // 3  # Calculate season based on month

# Create a new DataFrame containing only numeric columns from X
numeric_data = animation_processed.select_dtypes(include=['float64', 'int64'])  # Select only numeric columns (float64 and int64)
# remove response var
numeric_data = numeric_data.drop(columns=['revenue'])
```

```
<ipython-input-52-45715a33d49f>:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-cc
  animation_processed['release_date'] = pd.to_datetime(animation_processed['release_date'])
<ipython-input-52-45715a33d49f>:7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-cc
  animation_processed['release_month'] = animation_processed['release_date'].dt.month
<ipython-input-52-45715a33d49f>:8: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-cc
  animation_processed['release_year'] = animation_processed['release_date'].dt.year
<ipython-input-52-45715a33d49f>:9: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

```
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-cc
  animation_processed['release_day_of_week'] = animation_processed['release_date'].dt.dayofweek  # Monday=0, Sunday=6
<ipython-input-52-45715a33d49f>:10: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-cc
  animation_processed['release_season'] = (animation_processed['release_month'] % 12 + 3) // 3  # Calculate season based on month
```

```
import matplotlib.pyplot as plt

num_cols = 3

num_rows = (len(numeric_data.columns) + num_cols - 1) // num_cols

fig, axs = plt.subplots(num_rows, num_cols, figsize=(15, 5*num_rows))

axs = axs.flatten()

for i, column in enumerate(numeric_data.columns):
    ax = axs[i]
    ax.hist(numeric_data[column], bins=10)
    ax.set_title(column)
    ax.set_xlabel('Value')
    ax.set_ylabel('Frequency')

plt.tight_layout()
plt.show()
```



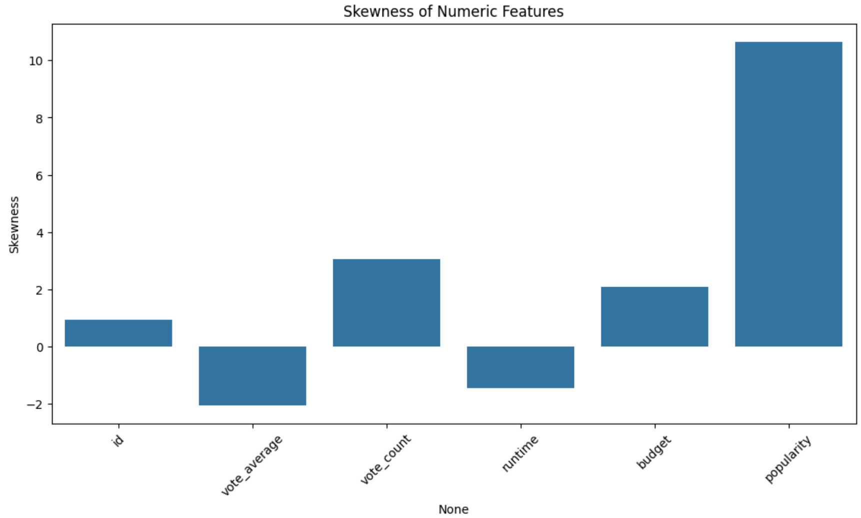
Degenerative Distributions

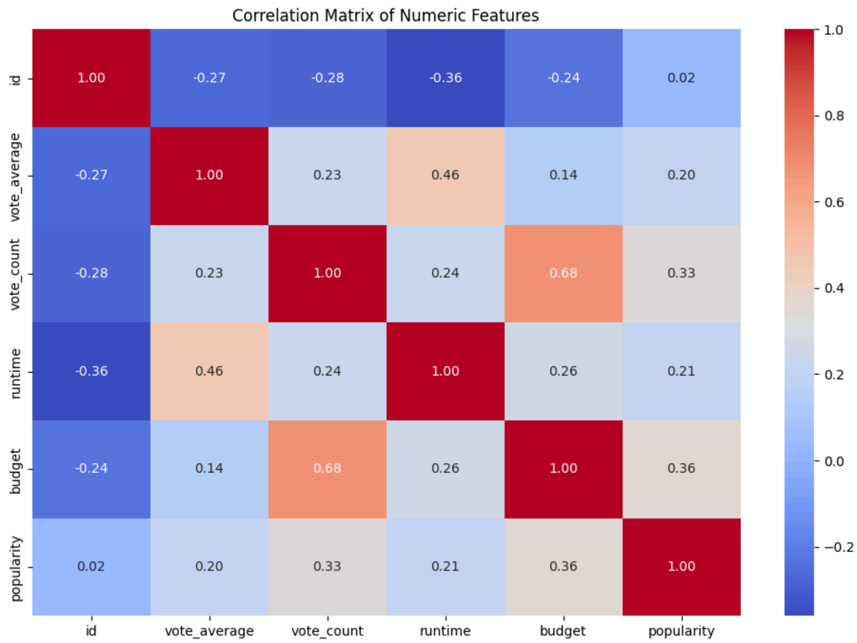
```
# Near Zero Variance
# Ratio of the frequency of the most prevalent value to the frequency of the second most prevalent is large ex: ratio of 20
import pandas as pd


# Function to check if both criteria are met
def is_near_zero_variance(series):
    unique_fraction = series.nunique() / len(series)
    value_counts = series.value_counts()
    if len(value_counts) > 1:
        freq_ratio = value_counts.iloc[0] / value_counts.iloc[1]
        if unique_fraction < 0.05 and freq_ratio > 20:
            return True
    return False

# Filter columns that meet the criteria
near_zero_variance_columns = []
for col in numeric_data.columns:
    if is_near_zero_variance(numeric_data[col]):
        near_zero_variance_columns.append(col)

# Print the indices or column names with near-zero variance
print("Columns with near-zero variance:")
print(near_zero_variance_columns)

# Drop columns with near-zero variance
#numeric_data.drop(columns=near_zero_variance_columns, inplace=True)
```

```
Columns with near-zero variance:
[]
```

```
# Need to Check for Outliers
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
# Check for outliers
def detect_outliers(data, threshold=3):
    z_scores = np.abs((data - data.mean()) / data.std())
    return (z_scores > threshold).any(axis=1)

outliers = detect_outliers(numeric_data)
print("Number of outliers:", outliers.sum())

# Check skewness
skewness = numeric_data.skew()
print("Skewness:")
print(skewness)

# Visualize skewness
plt.figure(figsize=(12, 6))
sns.barplot(x=skewness.index, y=skewness.values)
plt.title('Skewness of Numeric Features')
plt.xticks(rotation=45)
plt.ylabel('Skewness')
plt.show()

# Check correlations
correlation_matrix = numeric_data.corr()

# Visualize correlation matrix
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix of Numeric Features')
plt.show()
```

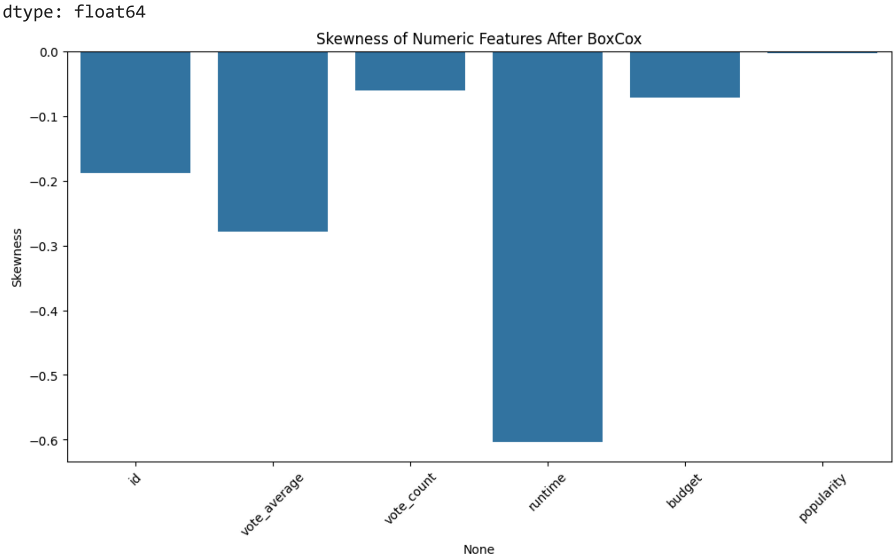
```
vote_average    -2.045109
vote_count       3.051876
runtime         -1.449774
budget           2.082128
popularity      10.652583
dtype: float64
```





```
from scipy.stats import boxcox

# Apply Box-Cox transformation to reduce skewness for all numeric features
for feature in numeric_data.columns:
    numeric_data[feature], _ = boxcox(numeric_data[feature] + 1)  # Adding 1 to handle zero values

# Recheck Skewness
skewness = numeric_data.skew()
print("Skewness:")
print(skewness)

# Visualize skewness
plt.figure(figsize=(12, 6))
sns.barplot(x=skewness.index, y=skewness.values)
plt.title('Skewness of Numeric Features After BoxCox')
plt.xticks(rotation=45)
plt.ylabel('Skewness')
plt.show()

# detect outliers after transformation
def detect_outliers(data, threshold=3):
    z_scores = np.abs((data - data.mean()) / data.std())
    return (z_scores > threshold).any(axis=1)

outliers = detect_outliers(numeric_data)
print("Number of outliers After Box Cox:", outliers.sum())
```

```
Skewness:
id             -0.187793
vote_average   -0.279025
vote_count     -0.061265
runtime        -0.603186
budget         -0.072158
popularity     -0.003517
dtype: float64
```




```
Number of outliers After Box Cox: 27
```

No predictors are highly skewed anymore Outliers were drastically reduced to only 24

```
# Compute the correlation matrix
correlation_matrix = numeric_data.corr()

# Print out highly correlated variables
highly_correlated_vars = correlation_matrix[(correlation_matrix > 0.75) & (correlation_matrix < 1.0)]
highly_correlated_vars = highly_correlated_vars.dropna(axis=1, how='all').dropna(axis=0, how='all')

print("Highly correlated variables:")
print(highly_correlated_vars)
```

```
Highly correlated variables:
            vote_count  popularity
vote_count         NaN    0.883102
popularity    0.883102         NaN
```

Remove one of these since its almost identical information

```
numeric_data.drop(columns=['popularity'], inplace=True)
```

```
animation_processed[ ['vote_average', 'vote_count', 'runtime', 'budget'] ] = numeric_data[['vote_average', 'vote_count', 'runtime', 'budget'
```

```
<ipython-input-59-b78b549e94b1>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-co
  animation_processed[ ['vote_average', 'vote_count', 'runtime', 'budget'] ] = numeric_data[['vote_average', 'vote_count', 'runtime', 'b
```

Text Preprocessing for BERT

```
from transformers import AutoTokenizer, BertModel
import torch

# Load pre-trained BERT model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("google-bert/bert-base-cased")
model = BertModel.from_pretrained("google-bert/bert-base-cased")

# Adjust batch size and sequence length
batch_size = 8
max_seq_length = 512

# Tokenize and process in batches
num_samples = len(animation_processed['overview'])
overview_embeddings = []

for i in range(0, num_samples, batch_size):
    batch_overviews = animation_processed['overview'][i:i+batch_size].tolist()
    tokenized_overviews = tokenizer(batch_overviews, max_length=max_seq_length, truncation=True, padding=True, return_tensors="pt")

    input_ids_batch = tokenized_overviews['input_ids']
    attention_mask_batch = tokenized_overviews['attention_mask']

    with torch.no_grad():
        outputs = model(input_ids_batch, attention_mask=attention_mask_batch)

    # Extract BERT embeddings
    bert_embeddings = outputs.last_hidden_state

    # Compute mean over sequence length dimension
    overview_embeddings_batch = torch.mean(bert_embeddings, dim=1)

    # Append to list
    overview_embeddings.append(overview_embeddings_batch)

# Concatenate embeddings from all batches
overview_embeddings = torch.cat(overview_embeddings, dim=0)

# Update animation_processed with embeddings
animation_processed['overview'] = overview_embeddings.numpy()
```

```
<ipython-input-60-82a8d8798fef>:39: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-cc
  animation_processed['overview'] = overview_embeddings.numpy()
```

Split the Data

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split


# Extract relevant features for prediction
X = animation_processed[['vote_average', 'vote_count', 'runtime', 'overview', 'release_month', 'release_year',
                         'release_day_of_week', 'release_season', 'budget']]
y = animation_processed['revenue']

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Scale the target variable
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()
```

```
!pip install torch
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (2.2.1+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch) (3.13.4)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (from torch) (4.11.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch) (3.3)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch) (3.1.3)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch) (2023.6.0)
Collecting nvidia-cuda-nvrtc-cu12==12.1.105 (from torch)
  Using cached nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (23.7 MB)
Collecting nvidia-cuda-runtime-cu12==12.1.105 (from torch)
  Using cached nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (823 kB)
Collecting nvidia-cuda-cupti-cu12==12.1.105 (from torch)
  Using cached nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (14.1 MB)
Collecting nvidia-cudnn-cu12==8.9.2.26 (from torch)
  Using cached nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl (731.7 MB)
Collecting nvidia-cublas-cu12==12.1.3.1 (from torch)
  Using cached nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl (410.6 MB)
Collecting nvidia-cufft-cu12==11.0.2.54 (from torch)
  Using cached nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl (121.6 MB)
Collecting nvidia-curand-cu12==10.3.2.106 (from torch)
  Using cached nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl (56.5 MB)
Collecting nvidia-cusolver-cu12==11.4.5.107 (from torch)
  Using cached nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_x86_64.whl (124.2 MB)
Collecting nvidia-cusparse-cu12==12.1.0.106 (from torch)
  Using cached nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_x86_64.whl (196.0 MB)
Collecting nvidia-nccl-cu12==2.19.3 (from torch)
  Using cached nvidia_nccl_cu12-2.19.3-py3-none-manylinux1_x86_64.whl (166.0 MB)
Collecting nvidia-nvtx-cu12==12.1.105 (from torch)
  Using cached nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (99 kB)
Requirement already satisfied: triton==2.2.0 in /usr/local/lib/python3.10/dist-packages (from torch) (2.2.0)
Collecting nvidia-nvjitlink-cu12 (from nvidia-cusolver-cu12==11.4.5.107->torch)
  Using cached nvidia_nvjitlink_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (21.1 MB)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch) (2.1.5)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-packages (from sympy->torch) (1.3.0)
Installing collected packages: nvidia-nvtx-cu12, nvidia-nvjitlink-cu12, nvidia-nccl-cu12, nvidia-curand-cu12, nvidia-cufft-cu12, nvidia-
Successfully installed nvidia-cublas-cu12-12.1.3.1 nvidia-cuda-cupti-cu12-12.1.105 nvidia-cuda-nvrtc-cu12-12.1.105 nvidia-cuda-runtime-c
```

Train Models

```
from sklearn.model_selection import GridSearchCV, KFold
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

def train_model(model, grid, x, y):
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    scoring = {'MAE': 'neg_mean_absolute_error',
               'MSE': 'neg_mean_squared_error',
               'R2': 'r2'}
    grid_search = GridSearchCV(estimator=model, param_grid=grid,
                               n_jobs=-1, cv=cv, scoring=scoring,
                               refit='R2', error_score=0)
    grid_result = grid_search.fit(x, y)
    best_model = grid_result.best_estimator_
    best_params = grid_result.best_params_

    # Training the best model using the entire dataset
    best_model.fit(x, y)

    # Predicting on the same dataset to compute training metrics
    y_pred = best_model.predict(x)

    n = len(y)  # Number of observations
    p = x.shape[1]  # Number of features
    r2 = r2_score(y, y_pred)
    adjusted_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)

    # Printing training metrics
    print("Training Metrics:")
    print("-----------------")
    print("Best R-squared on training data: %f using the following parameters: %s" % (r2_score(y, y_pred), best_params))
    print("Best Adjusted R-squared on training data: %f" % adjusted_r2)
    print("Mean Absolute Error on training data: %f" % mean_absolute_error(y, y_pred))
    print("Mean Squared Error on training data: %f" % mean_squared_error(y, y_pred))

    # Additional information: printing cross-validation results
    print("\nCross-Validation Metrics:")
    print("-------------------------")
    means = grid_result.cv_results_['mean_test_R2']
    stds = grid_result.cv_results_['std_test_R2']
    params = grid_result.cv_results_['params']
    for mean, stdev, param in zip(means, stds, params):
        print("R-squared: %f (%f) with parameters: %s" % (mean, stdev, param))
    return best_model
# Example usage:
# train_model(your_model, your_grid, your_X, your_y)
```

```
#Linear Regression
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import GridSearchCV
model = LinearRegression()
grid = {}
train_model(model, grid, X_train_scaled, y_train_scaled)
```

```
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/backend/fork_exec.py:38: F
  pid = os.fork()
Training Metrics:
-----------------
Best R-squared on training data: 0.444849 using the following parameters: {}
Best Adjusted R-squared on training data: 0.438658
Mean Absolute Error on training data: 0.491393
Mean Squared Error on training data: 0.555151

Cross-Validation Metrics:
-------------------------
R-squared: 0.423529 (0.059571) with parameters: {}
▾ LinearRegression
LinearRegression()
```

```
model = LinearRegression()

# Fit the model with the training data
model.fit(X_train_scaled, y_train_scaled)

# Make predictions on the test data
pred = model.predict(X_test_scaled)

print("Testing Metrics:")
print("-----------------")
print("Best R-squared on testing data: %f " % (r2_score(y_test_scaled, pred)))
print("Mean Absolute Error on testing data: %f" % mean_absolute_error(y_test_scaled, pred))
print("Mean Squared Error on testing data: %f" % mean_squared_error(y_test_scaled, pred))
```

```
Testing Metrics:
-----------------
Best R-squared on testing data: 0.362976
Mean Absolute Error on testing data: 0.582805
Mean Squared Error on testing data: 0.844603
```

```
#SVM
from sklearn.svm import SVR
model = SVR()
grid = {}
train_model(model, grid, X_train_scaled, y_train_scaled)
```

```
Training Metrics:
-----------------
Best R-squared on training data: 0.725812 using the following parameters: {}
Best Adjusted R-squared on training data: 0.722754
Mean Absolute Error on training data: 0.226368
Mean Squared Error on training data: 0.274188

Cross-Validation Metrics:
-------------------------
R-squared: 0.672617 (0.085246) with parameters: {}
```

▾ SVR
SVR()

```
model = SVR()

# Define the hyperparameter grid for tuning
param_grid = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'rbf'],
    'gamma': ['scale', 'auto']
}

# Train the model using the train_model function
best_model = train_model(model, param_grid, X_train_scaled, y_train_scaled)

# Make predictions on the test set using the best model
y_pred_test = best_model.predict(X_test_scaled)

print("\nTesting Metrics:")
print("-----------------")
print("R-squared on testing data: %f" % r2_score(y_test_scaled, y_pred_test))
print("Mean Absolute Error on testing data: %f" % mean_absolute_error(y_test_scaled, y_pred_test))
print("Mean Squared Error on testing data: %f" % mean_squared_error(y_test_scaled, y_pred_test))
```

```
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/backend/fork_exec.py:38: RuntimeWarning: os.fork() was called. os.fork() i
  pid = os.fork()
Training Metrics:
-----------------
Best R-squared on training data: 0.850757 using the following parameters: {'C': 10, 'gamma': 'auto', 'kernel': 'rbf'}
Best Adjusted R-squared on training data: 0.849092
Mean Absolute Error on training data: 0.172031
Mean Squared Error on training data: 0.149243

Cross-Validation Metrics:
-------------------------
R-squared: 0.267536 (0.061148) with parameters: {'C': 0.1, 'gamma': 'scale', 'kernel': 'linear'}
R-squared: 0.464732 (0.078891) with parameters: {'C': 0.1, 'gamma': 'scale', 'kernel': 'rbf'}
R-squared: 0.267536 (0.061148) with parameters: {'C': 0.1, 'gamma': 'auto', 'kernel': 'linear'}
R-squared: 0.464752 (0.078864) with parameters: {'C': 0.1, 'gamma': 'auto', 'kernel': 'rbf'}
R-squared: 0.269992 (0.061268) with parameters: {'C': 1, 'gamma': 'scale', 'kernel': 'linear'}
R-squared: 0.672617 (0.085246) with parameters: {'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}
R-squared: 0.269992 (0.061268) with parameters: {'C': 1, 'gamma': 'auto', 'kernel': 'linear'}
```

```
R-squared: 0.672644 (0.085230) with parameters: {'C': 1, 'gamma': 'auto', 'kernel': 'rbf'}
R-squared: 0.270182 (0.061164) with parameters: {'C': 10, 'gamma': 'scale', 'kernel': 'linear'}
R-squared: 0.705110 (0.079087) with parameters: {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}
R-squared: 0.270182 (0.061164) with parameters: {'C': 10, 'gamma': 'auto', 'kernel': 'linear'}
R-squared: 0.705144 (0.079164) with parameters: {'C': 10, 'gamma': 'auto', 'kernel': 'rbf'}

Testing Metrics:
-----------------
R-squared on testing data: 0.641484
Mean Absolute Error on testing data: 0.329119
Mean Squared Error on testing data: 0.475342
```

```
#Decision Tree
from sklearn.tree import DecisionTreeRegressor
import numpy as np
model = DecisionTreeRegressor()
grid = {}
train_model(model, grid, X_train_scaled, y_train_scaled)
```

```
Training Metrics:
-----------------
Best R-squared on training data: 1.000000 using the following parameters: {}
Best Adjusted R-squared on training data: 1.000000
Mean Absolute Error on training data: 0.000000
Mean Squared Error on training data: 0.000000

Cross-Validation Metrics:
-------------------------
R-squared: 0.510989 (0.185972) with parameters: {}
▾ DecisionTreeRegressor
DecisionTreeRegressor()
```

```
model = DecisionTreeRegressor(random_state=42)

# Define the hyperparameter grid for tuning
param_grid = {
    'max_depth': [None, 5, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Train the model using the train_model function
best_model = train_model(model, param_grid, X_train_scaled, y_train_scaled)

# Make predictions on the test set using the best model
y_pred_test = best_model.predict(X_test_scaled)

print("\nTesting Metrics:")
print("-----------------")
print("R-squared on testing data: %f" % r2_score(y_test_scaled, y_pred_test))
print("Mean Absolute Error on testing data: %f" % mean_absolute_error(y_test_scaled, y_pred_test))
print("Mean Squared Error on testing data: %f" % mean_squared_error(y_test_scaled, y_pred_test))
```

```
Training Metrics:
-----------------
Best R-squared on training data: 0.856389 using the following parameters: {'max_depth': 5, 'min_samples_leaf': 4, 'min_samples_split': 2
Best Adjusted R-squared on training data: 0.854787
Mean Absolute Error on training data: 0.184972
Mean Squared Error on training data: 0.143611

Cross-Validation Metrics:
-------------------------
R-squared: 0.544462 (0.133195) with parameters: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2}
R-squared: 0.585489 (0.112231) with parameters: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 5}
R-squared: 0.618603 (0.097159) with parameters: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 10}
R-squared: 0.600433 (0.165373) with parameters: {'max_depth': None, 'min_samples_leaf': 2, 'min_samples_split': 2}
R-squared: 0.616618 (0.165603) with parameters: {'max_depth': None, 'min_samples_leaf': 2, 'min_samples_split': 5}
R-squared: 0.619880 (0.109172) with parameters: {'max_depth': None, 'min_samples_leaf': 2, 'min_samples_split': 10}
R-squared: 0.625125 (0.124574) with parameters: {'max_depth': None, 'min_samples_leaf': 4, 'min_samples_split': 2}
R-squared: 0.625125 (0.124574) with parameters: {'max_depth': None, 'min_samples_leaf': 4, 'min_samples_split': 5}
R-squared: 0.612312 (0.121488) with parameters: {'max_depth': None, 'min_samples_leaf': 4, 'min_samples_split': 10}
R-squared: 0.626348 (0.098093) with parameters: {'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 2}
R-squared: 0.594393 (0.129044) with parameters: {'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 5}
R-squared: 0.625542 (0.091212) with parameters: {'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 10}
R-squared: 0.624023 (0.120872) with parameters: {'max_depth': 5, 'min_samples_leaf': 2, 'min_samples_split': 2}
R-squared: 0.628149 (0.116057) with parameters: {'max_depth': 5, 'min_samples_leaf': 2, 'min_samples_split': 5}
```

```
R-squared: 0.631638 (0.111746) with parameters: {'max_depth': 5, 'min_samples_leaf': 2, 'min_samples_split': 10}
R-squared: 0.638832 (0.095079) with parameters: {'max_depth': 5, 'min_samples_leaf': 4, 'min_samples_split': 2}
R-squared: 0.638832 (0.095079) with parameters: {'max_depth': 5, 'min_samples_leaf': 4, 'min_samples_split': 5}
R-squared: 0.625782 (0.101387) with parameters: {'max_depth': 5, 'min_samples_leaf': 4, 'min_samples_split': 10}
R-squared: 0.566145 (0.129255) with parameters: {'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 2}
R-squared: 0.591112 (0.121279) with parameters: {'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 5}
R-squared: 0.621680 (0.094724) with parameters: {'max_depth': 10, 'min_samples_leaf': 1, 'min_samples_split': 10}
R-squared: 0.604006 (0.167686) with parameters: {'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 2}
R-squared: 0.632117 (0.160008) with parameters: {'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 5}
R-squared: 0.631129 (0.105668) with parameters: {'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 10}
R-squared: 0.632461 (0.108437) with parameters: {'max_depth': 10, 'min_samples_leaf': 4, 'min_samples_split': 2}
R-squared: 0.632461 (0.108437) with parameters: {'max_depth': 10, 'min_samples_leaf': 4, 'min_samples_split': 5}
R-squared: 0.612058 (0.121517) with parameters: {'max_depth': 10, 'min_samples_leaf': 4, 'min_samples_split': 10}
R-squared: 0.577196 (0.137590) with parameters: {'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 2}
R-squared: 0.576274 (0.117435) with parameters: {'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 5}
R-squared: 0.585336 (0.132207) with parameters: {'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 10}
R-squared: 0.598360 (0.167245) with parameters: {'max_depth': 20, 'min_samples_leaf': 2, 'min_samples_split': 2}
R-squared: 0.614825 (0.164011) with parameters: {'max_depth': 20, 'min_samples_leaf': 2, 'min_samples_split': 5}
R-squared: 0.629290 (0.106767) with parameters: {'max_depth': 20, 'min_samples_leaf': 2, 'min_samples_split': 10}
R-squared: 0.625125 (0.124574) with parameters: {'max_depth': 20, 'min_samples_leaf': 4, 'min_samples_split': 2}
R-squared: 0.625125 (0.124574) with parameters: {'max_depth': 20, 'min_samples_leaf': 4, 'min_samples_split': 5}
R-squared: 0.612312 (0.121488) with parameters: {'max_depth': 20, 'min_samples_leaf': 4, 'min_samples_split': 10}

Testing Metrics:
-----------------
R-squared on testing data: 0.583116
Mean Absolute Error on testing data: 0.318899
Mean Squared Error on testing data: 0.552729
```

```
#KNN
from sklearn.neighbors import KNeighborsRegressor
grid = {
    'n_neighbors': [1,2,3,4,5,6,7,8,9,10,11,12],
    'weights': ['uniform', 'distance'],
    'p': [1,2] # p=1 compute manhattan distance, p=2 compute euclidean distance
    }
model = KNeighborsRegressor()
train_model(model, grid, X_train_scaled, y_train_scaled)
```

```
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/backend/fork_exec.py:38
  pid = os.fork()
Training Metrics:
-----------------
Best R-squared on training data: 0.710321 using the following parameters: {'n_neighbo
Best Adjusted R-squared on training data: 0.707090
Mean Absolute Error on training data: 0.256568
Mean Squared Error on training data: 0.289679

Cross-Validation Metrics:
-------------------------
R-squared: 0.055739 (0.407113) with parameters: {'n_neighbors': 1, 'p': 1, 'weights'
R-squared: 0.055739 (0.407113) with parameters: {'n_neighbors': 1, 'p': 1, 'weights'
R-squared: 0.128635 (0.400240) with parameters: {'n_neighbors': 1, 'p': 2, 'weights'
R-squared: 0.128635 (0.400240) with parameters: {'n_neighbors': 1, 'p': 2, 'weights'
R-squared: 0.346195 (0.240547) with parameters: {'n_neighbors': 2, 'p': 1, 'weights'
R-squared: 0.339686 (0.237778) with parameters: {'n_neighbors': 2, 'p': 1, 'weights'
R-squared: 0.363979 (0.182226) with parameters: {'n_neighbors': 2, 'p': 2, 'weights'
R-squared: 0.358909 (0.189520) with parameters: {'n_neighbors': 2, 'p': 2, 'weights'
R-squared: 0.503670 (0.158588) with parameters: {'n_neighbors': 3, 'p': 1, 'weights'
R-squared: 0.485198 (0.172821) with parameters: {'n_neighbors': 3, 'p': 1, 'weights'
R-squared: 0.499503 (0.129007) with parameters: {'n_neighbors': 3, 'p': 2, 'weights'
R-squared: 0.481366 (0.137049) with parameters: {'n_neighbors': 3, 'p': 2, 'weights'
R-squared: 0.523837 (0.146237) with parameters: {'n_neighbors': 4, 'p': 1, 'weights'
R-squared: 0.511389 (0.150959) with parameters: {'n_neighbors': 4, 'p': 1, 'weights'
R-squared: 0.503426 (0.131465) with parameters: {'n_neighbors': 4, 'p': 2, 'weights'
R-squared: 0.493437 (0.143602) with parameters: {'n_neighbors': 4, 'p': 2, 'weights'
R-squared: 0.545747 (0.164019) with parameters: {'n_neighbors': 5, 'p': 1, 'weights'
R-squared: 0.535853 (0.162874) with parameters: {'n_neighbors': 5, 'p': 1, 'weights'
R-squared: 0.503263 (0.149431) with parameters: {'n_neighbors': 5, 'p': 2, 'weights'
R-squared: 0.499121 (0.148701) with parameters: {'n_neighbors': 5, 'p': 2, 'weights'
R-squared: 0.555494 (0.132343) with parameters: {'n_neighbors': 6, 'p': 1, 'weights'
R-squared: 0.549979 (0.133821) with parameters: {'n_neighbors': 6, 'p': 1, 'weights'
R-squared: 0.528230 (0.128393) with parameters: {'n_neighbors': 6, 'p': 2, 'weights'
R-squared: 0.524784 (0.132933) with parameters: {'n_neighbors': 6, 'p': 2, 'weights'
R-squared: 0.565956 (0.100144) with parameters: {'n_neighbors': 7, 'p': 1, 'weights'
R-squared: 0.563464 (0.105999) with parameters: {'n_neighbors': 7, 'p': 1, 'weights'
R-squared: 0.528596 (0.131381) with parameters: {'n_neighbors': 7, 'p': 2, 'weights'
R-squared: 0.529031 (0.131957) with parameters: {'n_neighbors': 7, 'p': 2, 'weights'
R-squared: 0.561307 (0.100410) with parameters: {'n_neighbors': 8, 'p': 1, 'weights'
R-squared: 0.561288 (0.104577) with parameters: {'n_neighbors': 8, 'p': 1, 'weights'
R-squared: 0.515856 (0.134235) with parameters: {'n_neighbors': 8, 'p': 2, 'weights'
R-squared: 0.520106 (0.134913) with parameters: {'n_neighbors': 8, 'p': 2, 'weights'
R-squared: 0.552582 (0.103692) with parameters: {'n_neighbors': 9, 'p': 1, 'weights'
R-squared: 0.556923 (0.104781) with parameters: {'n_neighbors': 9, 'p': 1, 'weights'
R-squared: 0.515618 (0.119110) with parameters: {'n_neighbors': 9, 'p': 2, 'weights'
R-squared: 0.520826 (0.120324) with parameters: {'n_neighbors': 9, 'p': 2, 'weights'
R-squared: 0.548638 (0.103768) with parameters: {'n_neighbors': 10, 'p': 1, 'weights
R-squared: 0.554051 (0.107215) with parameters: {'n_neighbors': 10, 'p': 1, 'weights
R-squared: 0.522742 (0.130543) with parameters: {'n_neighbors': 10, 'p': 2, 'weights
R-squared: 0.528292 (0.127265) with parameters: {'n_neighbors': 10, 'p': 2, 'weights
R-squared: 0.557539 (0.109444) with parameters: {'n_neighbors': 11, 'p': 1, 'weights
R-squared: 0.562758 (0.108695) with parameters: {'n_neighbors': 11, 'p': 1, 'weights
R-squared: 0.528249 (0.118602) with parameters: {'n_neighbors': 11, 'p': 2, 'weights
R-squared: 0.533461 (0.119529) with parameters: {'n_neighbors': 11, 'p': 2, 'weights
R-squared: 0.550450 (0.114516) with parameters: {'n_neighbors': 12, 'p': 1, 'weights
```

```
model = KNeighborsRegressor()

# Define the hyperparameter grid for tuning
param_grid = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'p': [1, 2]
}

# Train the model using the train_model function
best_model = train_model(model, param_grid, X_train_scaled, y_train_scaled)

# Make predictions on the test set using the best model
y_pred_test = best_model.predict(X_test_scaled)

print("\nTesting Metrics:")
print("-----------------")
print("R-squared on testing data: %f" % r2_score(y_test_scaled, y_pred_test))
print("Mean Absolute Error on testing data: %f" % mean_absolute_error(y_test_scaled, y_pred_test))
print("Mean Squared Error on testing data: %f" % mean_squared_error(y_test_scaled, y_pred_test))
```

```
Training Metrics:
-----------------
Best R-squared on training data: 1.000000 using the following parameters: {'n_neighbors': 9, 'p': 1, 'weights': 'distance'}
Best Adjusted R-squared on training data: 1.000000
Mean Absolute Error on training data: 0.000000
Mean Squared Error on training data: 0.000000

Cross-Validation Metrics:
-------------------------
R-squared: 0.589690 (0.078795) with parameters: {'n_neighbors': 3, 'p': 1, 'weights': 'uniform'}
R-squared: 0.608761 (0.080879) with parameters: {'n_neighbors': 3, 'p': 1, 'weights': 'distance'}
R-squared: 0.520426 (0.120655) with parameters: {'n_neighbors': 3, 'p': 2, 'weights': 'uniform'}
R-squared: 0.538469 (0.125866) with parameters: {'n_neighbors': 3, 'p': 2, 'weights': 'distance'}
R-squared: 0.610038 (0.105527) with parameters: {'n_neighbors': 5, 'p': 1, 'weights': 'uniform'}
R-squared: 0.635747 (0.086876) with parameters: {'n_neighbors': 5, 'p': 1, 'weights': 'distance'}
R-squared: 0.553843 (0.143890) with parameters: {'n_neighbors': 5, 'p': 2, 'weights': 'uniform'}
R-squared: 0.576340 (0.141880) with parameters: {'n_neighbors': 5, 'p': 2, 'weights': 'distance'}
R-squared: 0.615799 (0.104769) with parameters: {'n_neighbors': 7, 'p': 1, 'weights': 'uniform'}
R-squared: 0.641272 (0.087417) with parameters: {'n_neighbors': 7, 'p': 1, 'weights': 'distance'}
R-squared: 0.570236 (0.110178) with parameters: {'n_neighbors': 7, 'p': 2, 'weights': 'uniform'}
R-squared: 0.596123 (0.104860) with parameters: {'n_neighbors': 7, 'p': 2, 'weights': 'distance'}
R-squared: 0.627171 (0.088082) with parameters: {'n_neighbors': 9, 'p': 1, 'weights': 'uniform'}
R-squared: 0.649674 (0.075068) with parameters: {'n_neighbors': 9, 'p': 1, 'weights': 'distance'}
R-squared: 0.589068 (0.096212) with parameters: {'n_neighbors': 9, 'p': 2, 'weights': 'uniform'}
R-squared: 0.612155 (0.090298) with parameters: {'n_neighbors': 9, 'p': 2, 'weights': 'distance'}

Testing Metrics:
-----------------
R-squared on testing data: 0.617165
Mean Absolute Error on testing data: 0.335526
Mean Squared Error on testing data: 0.507585
```

```
#Random forest
from sklearn.ensemble import RandomForestRegressor
grid ={}
model = RandomForestRegressor(n_estimators=200,random_state=42,max_depth=10)
RF_trained_model = train_model(model, grid, X_train_scaled, y_train_scaled)
```

```
Training Metrics:
-----------------
Best R-squared on training data: 0.962985 using the following parameters: {}
Best Adjusted R-squared on training data: 0.962573
Mean Absolute Error on training data: 0.095491
Mean Squared Error on training data: 0.037015

Cross-Validation Metrics:
-------------------------
R-squared: 0.734807 (0.115112) with parameters: {}
```

```
# Random Forest is best, try on test set
y_pred_RF = RF_trained_model.predict(X_test_scaled)
# Printing Testing metrics
print("Testing Metrics:")
print("-----------------")
print("Best R-squared on testing data: %f " % (r2_score(y_test_scaled, y_pred_RF)))
print("Mean Absolute Error on testing data: %f" % mean_absolute_error(y_test_scaled, y_pred_RF))
print("Mean Squared Error on testing data: %f" % mean_squared_error(y_test_scaled, y_pred_RF))
```

```
Testing Metrics:
-----------------
Best R-squared on testing data: 0.794337
Mean Absolute Error on testing data: 0.223626
Mean Squared Error on testing data: 0.272680
```

```
from sklearn.linear_model import Ridge
model = Ridge()
grid = {}
train_model(model, grid, X_train_scaled, y_train_scaled)
```

```
Training Metrics:
-----------------
Best R-squared on training data: 0.418998 using the following parameters: {}
Best Adjusted R-squared on training data: 0.412519
Mean Absolute Error on training data: 0.495713
Mean Squared Error on training data: 0.581002

Cross-Validation Metrics:
-------------------------
R-squared: 0.382804 (0.082189) with parameters: {}
▾ Ridge
Ridge()
```

```
from sklearn.linear_model import Lasso
model = Lasso()
grid={}
train_model(model, grid, X_train_scaled, y_train_scaled)
```

```
Training Metrics:
-----------------
Best R-squared on training data: 0.000000 using the following parameters: {}
Best Adjusted R-squared on training data: -0.011152
Mean Absolute Error on training data: 0.628581
Mean Squared Error on training data: 1.000000

Cross-Validation Metrics:
-------------------------
R-squared: -0.016464 (0.010604) with parameters: {}
▾ Lasso
Lasso()
```

```
# Elastic Net
grid = {
    'alpha': [0.1, 0.5, 1.0],  # Regularization strength
    'l1_ratio': [0.1, 0.5, 0.7, 0.9]  # Mixing parameter, with 0 <= l1_ratio <= 1.
}
from sklearn.linear_model import ElasticNet
model = ElasticNet()

train_model(model, grid, X_train_scaled, y_train_scaled)
```

```
Training Metrics:
-----------------
Best R-squared on training data: 0.411573 using the following parameters: {'alpha': 0.1,
Best Adjusted R-squared on training data: 0.405011
Mean Absolute Error on training data: 0.481547
Mean Squared Error on training data: 0.588427

Cross-Validation Metrics:
-------------------------
R-squared: 0.389291 (0.070769) with parameters: {'alpha': 0.1, 'l1_ratio': 0.1}
R-squared: 0.384455 (0.071440) with parameters: {'alpha': 0.1, 'l1_ratio': 0.5}
R-squared: 0.380918 (0.074028) with parameters: {'alpha': 0.1, 'l1_ratio': 0.7}
R-squared: 0.375439 (0.077530) with parameters: {'alpha': 0.1, 'l1_ratio': 0.9}
R-squared: 0.361181 (0.073236) with parameters: {'alpha': 0.5, 'l1_ratio': 0.1}
R-squared: 0.281524 (0.073356) with parameters: {'alpha': 0.5, 'l1_ratio': 0.5}
R-squared: 0.216677 (0.064027) with parameters: {'alpha': 0.5, 'l1_ratio': 0.7}
R-squared: 0.150973 (0.055263) with parameters: {'alpha': 0.5, 'l1_ratio': 0.9}
R-squared: 0.298431 (0.073253) with parameters: {'alpha': 1.0, 'l1_ratio': 0.1}
R-squared: 0.069260 (0.036726) with parameters: {'alpha': 1.0, 'l1_ratio': 0.5}
R-squared: -0.016464 (0.010604) with parameters: {'alpha': 1.0, 'l1_ratio': 0.7}
R-squared: -0.016464 (0.010604) with parameters: {'alpha': 1.0, 'l1_ratio': 0.9}
▾          ElasticNet
ElasticNet(alpha=0.1, l1_ratio=0.1)
```

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Define the model architecture
class ANNModel(nn.Module):
    def __init__(self, input_size):
        super(ANNModel, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 1)
        )

    def forward(self, x):
        return self.layers(x)

# Convert data to PyTorch tensors
X_tensor = torch.from_numpy(X_train_scaled).float()
y_tensor = torch.from_numpy(y_train_scaled.reshape(-1, 1)).float()

# Define the training function
def train_model(model, X_tensor, y_tensor, epochs, batch_size, learning_rate, weight_decay):
    # Create DataLoader for training data
    train_dataset = TensorDataset(X_tensor, y_tensor)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

    # Set the loss function and optimizer
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay)

    # Training loop
    for epoch in range(epochs):
        model.train()
        train_loss = 0.0

        for batch_X, batch_y in train_loader:
            optimizer.zero_grad()
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()

        train_loss /= len(train_loader)

        if (epoch + 1) % 10 == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Train Loss: {train_loss:.4f}")

    return model

# Define the evaluation function
def evaluate_model(model, X_tensor, y_tensor):
    model.eval()
    with torch.no_grad():
        outputs = model(X_tensor)
        y_pred = outputs.numpy().flatten()
        y_true = y_tensor.numpy().flatten()

        mse = mean_squared_error(y_true, y_pred)
        mae = mean_absolute_error(y_true, y_pred)
        r2 = r2_score(y_true, y_pred)

        return mse, mae, r2

# Define the cross-validation function
def cross_validate(model, X_tensor, y_tensor, epochs, batch_size, learning_rate, weight_decay, n_splits):
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
```

```
    mse_scores = []
    mae_scores = []
    r2_scores = []

    for train_index, val_index in kf.split(X_tensor):
        X_train, X_val = X_tensor[train_index], X_tensor[val_index]
        y_train, y_val = y_tensor[train_index], y_tensor[val_index]

        model = ANNModel(X_tensor.shape[1])
        model = train_model(model, X_train, y_train, epochs, batch_size, learning_rate, weight_decay)

        mse, mae, r2 = evaluate_model(model, X_val, y_val)
        mse_scores.append(mse)
        mae_scores.append(mae)
        r2_scores.append(r2)

    return np.mean(mse_scores), np.mean(mae_scores), np.mean(r2_scores)

# Hyperparameter tuning
epochs = 100
batch_size = 32
learning_rate = 0.001
weight_decay = 0.01
n_splits = 5

# Perform cross-validation
mse, mae, r2 = cross_validate(ANNModel, X_tensor, y_tensor, epochs, batch_size, learning_rate, weight_decay, n_splits)

print(f"Cross-Validation Metrics:")
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"Mean Absolute Error (MAE): {mae:.4f}")
print(f"R-squared (R2): {r2:.4f}")

# Train the final model using the best hyperparameters
model = ANNModel(X_tensor.shape[1])
model = train_model(model, X_tensor, y_tensor, epochs, batch_size, learning_rate, weight_decay)

# Evaluate the final model on the test set
X_test_tensor = torch.from_numpy(X_test_scaled).float()
y_test_tensor = torch.from_numpy(y_test_scaled.reshape(-1, 1)).float()

mse, mae, r2 = evaluate_model(model, X_test_tensor, y_test_tensor)

print(f"\nFinal Model Metrics on Test Set:")
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"Mean Absolute Error (MAE): {mae:.4f}")
print(f"R-squared (R2): {r2:.4f}")
```

```
Epoch [20/100], Train Loss: 0.2450
Epoch [30/100], Train Loss: 0.2451
Epoch [40/100], Train Loss: 0.2370
Epoch [50/100], Train Loss: 0.1750
Epoch [60/100], Train Loss: 0.2288
Epoch [70/100], Train Loss: 0.1787
Epoch [80/100], Train Loss: 0.1625
Epoch [90/100], Train Loss: 0.1441
Epoch [100/100], Train Loss: 0.1544
Epoch [10/100], Train Loss: 0.3121
Epoch [20/100], Train Loss: 0.2601
Epoch [30/100], Train Loss: 0.2388
Epoch [40/100], Train Loss: 0.2051
Epoch [50/100], Train Loss: 0.2270
Epoch [60/100], Train Loss: 0.2107
Epoch [70/100], Train Loss: 0.2007
Epoch [80/100], Train Loss: 0.1951
Epoch [90/100], Train Loss: 0.1689
Epoch [100/100], Train Loss: 0.1630
Epoch [10/100], Train Loss: 0.3028
Epoch [20/100], Train Loss: 0.2411
```

```
Epoch [40/100], Train Loss: 0.2349
Epoch [50/100], Train Loss: 0.1881
Epoch [60/100], Train Loss: 0.1818
Epoch [70/100], Train Loss: 0.1546
Epoch [80/100], Train Loss: 0.1536
Epoch [90/100], Train Loss: 0.1429
Epoch [100/100], Train Loss: 0.1619
Cross-Validation Metrics:
Mean Squared Error (MSE): 0.2685
Mean Absolute Error (MAE): 0.2419
R-squared (R2): 0.7248
Epoch [10/100], Train Loss: 0.3007
Epoch [20/100], Train Loss: 0.2798
Epoch [30/100], Train Loss: 0.2138
Epoch [40/100], Train Loss: 0.2434
Epoch [50/100], Train Loss: 0.2594
Epoch [60/100], Train Loss: 0.1996
Epoch [70/100], Train Loss: 0.1903
Epoch [80/100], Train Loss: 0.1898
Epoch [90/100], Train Loss: 0.1719
Epoch [100/100], Train Loss: 0.1880

Final Model Metrics on Test Set:
Mean Squared Error (MSE): 0.3654
Mean Absolute Error (MAE): 0.2713
R-squared (R2): 0.7244
```

```
from datetime import datetime, timedelta


start_date = datetime(2024, 1, 1)
end_date = datetime(2024, 12, 31)

results_df = pd.DataFrame(columns=['sample_index', 'max_revenue_date', 'max_revenue_value'])


for index, row in X_test.iterrows():
    vote_average = row['vote_average']
    vote_count = row['vote_count']
    runtime = row['runtime']
    overview = row['overview']
    budget = row['budget']

    max_revenue_date = None
    max_revenue_value = float('-inf')

    current_date = start_date
    while current_date <= end_date:

        input_features = pd.DataFrame({
            'vote_average': [vote_average],
            'vote_count': [vote_count],
            'runtime': [runtime],
            'overview': [overview],
            'release_month': [current_date.month],
            'release_year': [current_date.year],
            'release_day_of_week': [current_date.weekday()],
            'release_season': [current_date.month % 12 // 3 + 1],
            'budget': [budget]
        })


        input_features_scaled = scaler.transform(input_features)



        predicted_revenue = RF_trained_model.predict(input_features_scaled)[0]


        if predicted_revenue > max_revenue_value:
            max_revenue_date = current_date
            max_revenue_value = predicted_revenue



        current_date += timedelta(days=1)

    # Inverse transform the predicted maximum revenue using the same scaler used for the target variable
    max_revenue_value = y_scaler.inverse_transform([[max_revenue_value]])[0][0]


    new_row = pd.DataFrame([{
    'sample_index': index,
    'max_revenue_date': max_revenue_date,
    'max_revenue_value': max_revenue_value
}])


    results_df = pd.concat([results_df,new_row], ignore_index=True)

print(results_df)
```

```
    sample_index max_revenue_date  max_revenue_value
0            858       2024-11-03       1.377930e+08
1           1318       2024-01-07       5.541671e+07
2           2892       2024-01-07       5.737219e+07
3            133       2024-11-07       2.123965e+08
4           1932       2024-01-07       8.630025e+07
..           ...              ...                ...
200          423       2024-11-04       1.197476e+08
201         2579       2024-01-07       5.760468e+07
202         2071       2024-12-01       1.281819e+08
203          669       2024-12-01       9.823075e+07
204        15026       2024-01-07       3.894095e+07
```

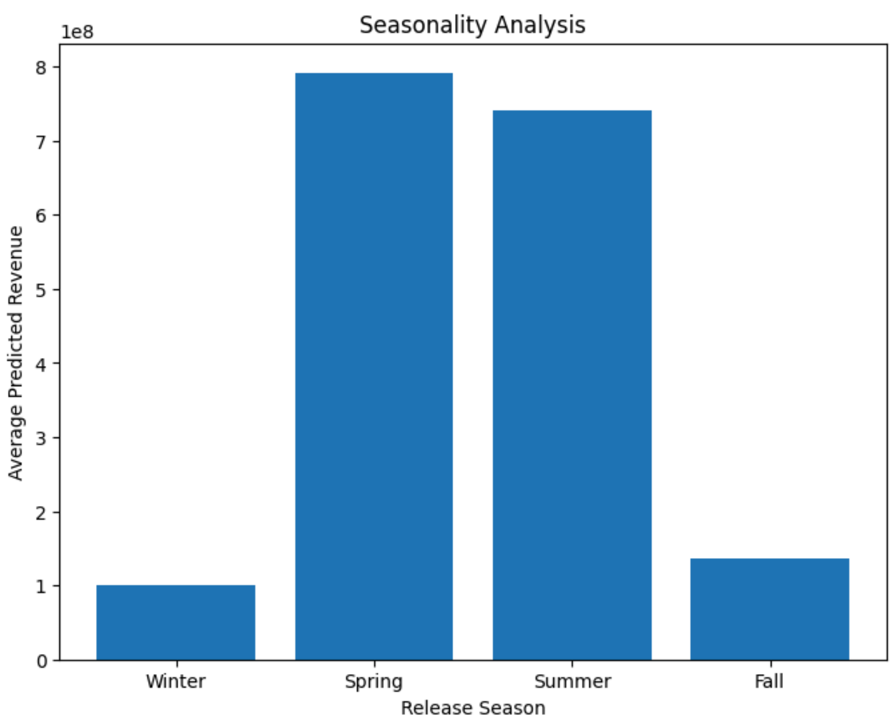
```
[205 rows x 3 columns]
```

```
animation_processed['release_season'].unique()
```

```
array([4, 1, 2, 3], dtype=int32)
```

```
# Extract the release season from the 'max_revenue_date'
results_df['release_season'] = results_df['max_revenue_date'].dt.month % 12 // 3 + 1

# Group by release season and calculate the average predicted revenue
seasonality_df = results_df.groupby('release_season')['max_revenue_value'].mean().reset_index()

# Print the seasonality results
print("Seasonality Analysis:")
print("---------------------")
print(seasonality_df)

# Visualize the seasonality using a bar plot
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))
plt.bar(seasonality_df['release_season'], seasonality_df['max_revenue_value'])
plt.xlabel('Release Season')
plt.ylabel('Average Predicted Revenue')
plt.title('Seasonality Analysis')
plt.xticks(seasonality_df['release_season'], ['Winter', 'Spring', 'Summer', 'Fall'])
plt.show()
```

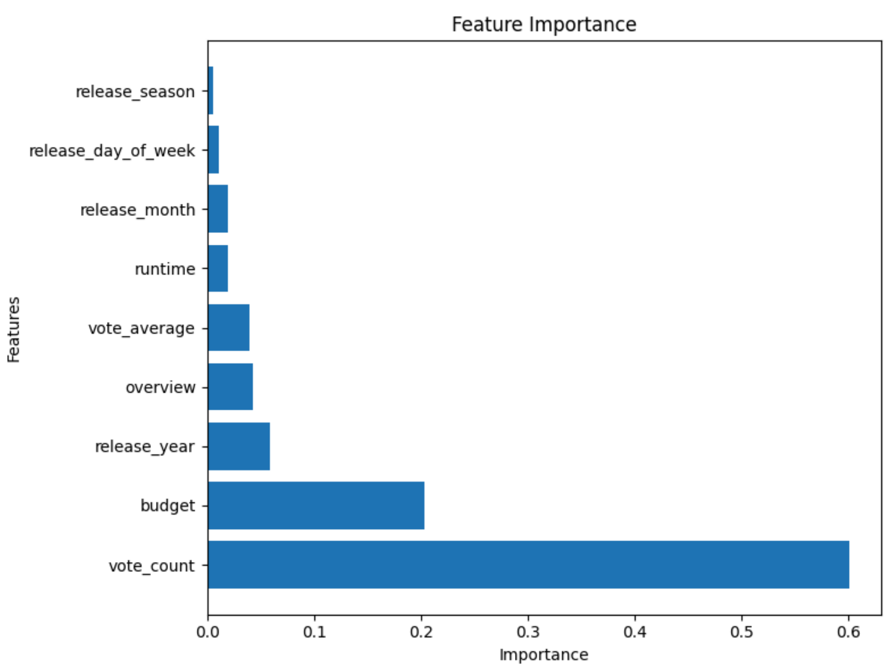
```
Seasonality Analysis:
---------------------
   release_season  max_revenue_value
0               1       1.007851e+08
1               2       7.917446e+08
2               3       7.411642e+08
3               4       1.356278e+08
```



```
from scipy.stats import f_oneway

winter_revenue = results_df[results_df['release_season'] == 1]['max_revenue_value']
spring_revenue = results_df[results_df['release_season'] == 2]['max_revenue_value']
summer_revenue = results_df[results_df['release_season'] == 3]['max_revenue_value']
fall_revenue = results_df[results_df['release_season'] == 4]['max_revenue_value']

f_statistic, p_value = f_oneway(winter_revenue, spring_revenue, summer_revenue, fall_revenue)

print("\nOne-way ANOVA Results:")
print("----------------------")
print(f"F-statistic: {f_statistic}")
print(f"p-value: {p_value}")
```

```
One-way ANOVA Results:
----------------------
F-statistic: 366.5983330269283
p-value: 3.25318403426854e-81
```

```
importances = RF_trained_model.feature_importances_

feature_names = X.columns

indices = np.argsort(importances)[::-1]

plt.figure(figsize=(8, 6))
plt.barh(range(len(importances)), importances[indices])
plt.yticks(range(len(importances)), feature_names[indices])
plt.xlabel('Importance')
plt.ylabel('Features')
plt.title('Feature Importance')
plt.tight_layout()
plt.savefig("Imp_feat.png")
plt.show()
```



Method 2

```
from datetime import datetime

animation_processed['release_date'] = pd.to_datetime(animation_processed['release_date'])

# Find the earliest date in the 'release_date' column
reference_date = animation_processed['release_date'].min()

animation_processed['days_since_ref'] = (animation_processed['release_date'] - reference_date).dt.days
animation_processed = animation_processed.drop('release_date', axis=1)
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split


# Extract relevant features for prediction
X = animation_processed[['vote_average', 'vote_count', 'runtime', 'overview', 'days_since_ref' ,'release_season', 'budget']]
y = animation_processed['revenue']


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Scale the target variable
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()
```

```
from sklearn.ensemble import RandomForestRegressor
grid ={}
model = RandomForestRegressor()
RF_trained_model = train_model(model, grid, X_train_scaled, y_train_scaled)
```

```
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/backend/fork_exec.py:38: RuntimeWarning: os.fork() was called. os.fork() i
  pid = os.fork()
Training Metrics:
-----------------
Best R-squared on training data: 0.959669 using the following parameters: {}
Best Adjusted R-squared on training data: 0.959320
Mean Absolute Error on training data: 0.088358
Mean Squared Error on training data: 0.040331

Cross-Validation Metrics:
-------------------------
R-squared: 0.669951 (0.262598) with parameters: {}
```

```
# Random Forest is best, try on test set
y_pred_RF = RF_trained_model.predict(X_test_scaled)
# Printing Testing metrics
print("Testing Metrics:")
print("-----------------")
print("Best R-squared on testing data: %f " % (r2_score(y_test_scaled, y_pred_RF)))
print("Mean Absolute Error on testing data: %f" % mean_absolute_error(y_test_scaled, y_pred_RF))
print("Mean Squared Error on testing data: %f" % mean_squared_error(y_test_scaled, y_pred_RF))
```

```
Testing Metrics:
-----------------
Best R-squared on testing data: 0.828233
Mean Absolute Error on testing data: 0.222037
Mean Squared Error on testing data: 0.195583
```

```
from datetime import datetime, timedelta

results_df = pd.DataFrame(columns=['sample_index', 'max_revenue_date', 'max_revenue_value'])

for index, row in X_test.iterrows():
    vote_average = row['vote_average']
    vote_count = row['vote_count']
    runtime = row['runtime']
    overview = row['overview']
    budget = row['budget']
    release_season = row['release_season']
    days_since_ref = row['days_since_ref']

    # Calculate the actual release date for the test sample
    release_date = reference_date + timedelta(days=days_since_ref)

    # Set the start and end dates based on the release date
    start_date = release_date
    end_date = release_date + timedelta(days=365)

    max_revenue_date = None
    max_revenue_value = float('-inf')

    current_date = start_date
    while current_date <= end_date:
        days_since_ref = (current_date - reference_date).days

        input_features = pd.DataFrame({
            'vote_average': [vote_average],
            'vote_count': [vote_count],
            'runtime': [runtime],
            'overview': [overview],
            'days_since_ref': [days_since_ref],
            'release_season': [release_season],
            'budget': [budget]
        })

        input_features_scaled = scaler.transform(input_features)
        predicted_revenue = RF_trained_model.predict(input_features_scaled)[0]

        if predicted_revenue > max_revenue_value:
            max_revenue_date = current_date
            max_revenue_value = predicted_revenue

        current_date += timedelta(days=1)

    # Inverse transform the predicted maximum revenue using the same scaler used for the target variable
    max_revenue_value = y_scaler.inverse_transform([[max_revenue_value]])[0][0]

    # Format the date as "mm-dd-yyyy"
    formatted_date = max_revenue_date.strftime("%m-%d-%Y")

    new_row = pd.DataFrame([{
        'sample_index': index,
        'max_revenue_date': formatted_date,
        'max_revenue_value': max_revenue_value
    }])

    results_df = pd.concat([results_df, new_row], ignore_index=True)

print(results_df)
```

```
     sample_index max_revenue_date  max_revenue_value
0             858       01-07-2023       1.385319e+08
1            1318       05-01-2012       6.852220e+06
2            2892       04-03-2017       3.400564e+06
3             133       11-08-1973       7.863896e+07
4            1932       12-17-2015       5.178023e+07
..            ...              ...                ...
200           423       10-06-2011       1.149347e+08
201          2579       01-11-2018       4.457915e+06
202          2071       01-24-2022       5.845061e+07
203           669       07-15-2014       3.901492e+07
204         15026       06-17-2018       1.537495e+04

[205 rows x 3 columns]
```

```
import matplotlib.pyplot as plt

# Extract the month from the 'max_revenue_date' column
results_df['max_revenue_month'] = pd.to_datetime(results_df['max_revenue_date']).dt.month

# Map the month numbers to season names
season_map = {
    12: 'Winter', 1: 'Winter', 2: 'Winter',
    3: 'Spring', 4: 'Spring', 5: 'Spring',
    6: 'Summer', 7: 'Summer', 8: 'Summer',
    9: 'Fall', 10: 'Fall', 11: 'Fall'
}

# Create a new column 'max_revenue_season' based on the 'max_revenue_month'
results_df['max_revenue_season'] = results_df['max_revenue_month'].map(season_map)

# Group by 'max_revenue_season' and calculate the mean 'max_revenue_value'
seasonality_df = results_df.groupby('max_revenue_season')['max_revenue_value'].mean().reset_index()

# Create a bar plot of the seasonality analysis
plt.figure(figsize=(8, 6))
plt.bar(seasonality_df['max_revenue_season'], seasonality_df['max_revenue_value'])
plt.xlabel('Season')
plt.ylabel('Average Maximum Revenue')
plt.title('Seasonality Analysis')
plt.show()
```