Forecasting Video Games Sales

Haolan Mai

Preparation and Data Cleaning

```
import pandas as pd;
import numpy as np;
import seaborn as sns
from matplotlib import pyplot as plt
from matplotlib import style
from sklearn import metrics
```

Reading and exploring data

```
data = pd.read_csv('videogame.csv')
data.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 16719 entries, 0 to 16718
## Data columns (total 16 columns):
##  #   Column           Non-Null Count  Dtype
## ---  ------           --------------  -----
##  0   Name             16717 non-null  object
##  1   Platform         16719 non-null  object
##  2   Year_of_Release  16450 non-null  float64
##  3   Genre            16717 non-null  object
##  4   Publisher        16665 non-null  object
##  5   NA_Sales         16719 non-null  float64
##  6   EU_Sales         16719 non-null  float64
##  7   JP_Sales         16719 non-null  float64
##  8   Other_Sales      16719 non-null  float64
##  9   Global_Sales     16719 non-null  float64
##  10  Critic_Score     8137 non-null   float64
##  11  Critic_Count     8137 non-null   float64
##  12  User_Score       10015 non-null  object
##  13  User_Count       7590 non-null   float64
##  14  Developer        10096 non-null  object
##  15  Rating           9950 non-null   object
## dtypes: float64(9), object(7)
## memory usage: 2.0+ MB
```

```
data.describe(include="all")
```

```
##                              Name Platform  ...  Developer Rating
## count                       16717    16719  ...      10096   9950
## unique                      11562       31  ...       1696      8
## top     Need for Speed: Most Wanted      PS2  ...    Ubisoft      E
## freq                           12     2161  ...        204   3991
## mean                          NaN      NaN  ...        NaN    NaN
## std                           NaN      NaN  ...        NaN    NaN
## min                           NaN      NaN  ...        NaN    NaN
## 25%                           NaN      NaN  ...        NaN    NaN
## 50%                           NaN      NaN  ...        NaN    NaN
## 75%                           NaN      NaN  ...        NaN    NaN
## max                           NaN      NaN  ...        NaN    NaN
##
## [11 rows x 16 columns]
```

There are ~17k games, but some of the data is missing. For instance, only around half of all games has a critic score. This might be a problem for the prediction model, as critic score can be one of the major factors determing the Global Sales. User_Score has non-numeric format. Most of the games have a pretty good score, 7+ (or 70+ for critic score)

In the next cell, I am doing 4 things:

Droping games without a year of release or genre Creating a new column for age of the game Converting User_Score to float and replacing the tbd value in dataset with NA

```
data = data.rename(columns={"Year_of_Release": "Year",
                            "NA_Sales": "NA",
                            "EU_Sales": "EU",
                            "JP_Sales": "JP",
                            "Other_Sales": "Other",
                            "Global_Sales": "Global"})
data = data[data["Year"].notnull()]
data = data[data["Genre"].notnull()]
data["Year"] = data["Year"].apply(int)
data["Age"] = 2018 - data["Year"]
data["User_Score"] = data["User_Score"].replace("tbd", np.nan).astype(float)
data.describe(include="all")
```

```
##                              Name Platform  ...  Rating           Age
## count                       16448    16448  ...    9769  16448.000000
## unique                      11429       31  ...       8           NaN
## top     Need for Speed: Most Wanted      PS2  ...       E           NaN
## freq                           12     2127  ...    3922           NaN
## mean                          NaN      NaN  ...     NaN     11.511004
## std                           NaN      NaN  ...     NaN      5.877470
## min                           NaN      NaN  ...     NaN     -2.000000
## 25%                           NaN      NaN  ...     NaN      8.000000
## 50%                           NaN      NaN  ...     NaN     11.000000
## 75%                           NaN      NaN  ...     NaN     15.000000
## max                           NaN      NaN  ...     NaN     38.000000
##
## [11 rows x 17 columns]
```

From, the output above, we can see: -

There are high outliers in sales columns (NA, EU, JP, Other, Global) and User_Count column.

They might be usefull for training as they indicate bestseller games, but for now I am going to remove them and maybe add them later. The below function can be used to remove outliers present in the data set. A data entry is called an outlier if: - value < Q1 - 3 * IQR value > Q3 + 3 * IQR

where,

Q1 - First Quartile Q3 - Thrid Quartile IQR - Inter-quartile range

```
def rm_outliers(df, list_of_keys):
    df_out = df
    for key in list_of_keys:

        # Calculate first and third quartile
        first_quartile = df_out[key].describe()["25%"]
        third_quartile = df_out[key].describe()["75%"]

        # Interquartile range
        iqr = third_quartile - first_quartile
        removed = df_out[(df_out[key] <= (first_quartile - 3 * iqr)) |
                        (df_out[key] >= (third_quartile + 3 * iqr))]
        df_out = df_out[(df_out[key] > (first_quartile - 3 * iqr)) &
                        (df_out[key] < (third_quartile + 3 * iqr))]

    return df_out, removed
```

```
data, rmvd_global = rm_outliers(data, ["Global"])
data.describe()
```

```
##                 Year            NA  ...    User_Count           Age
## count  15401.000000  15401.000000  ...   6747.000000  15401.000000
## mean    2006.592624      0.144688  ...    111.325033     11.407376
## std        5.758078      0.210709  ...    406.635191      5.758078
## min     1980.000000      0.000000  ...      4.000000     -2.000000
## 25%     2003.000000      0.000000  ...      9.000000      8.000000
## 50%     2007.000000      0.070000  ...     21.000000     11.000000
## 75%     2010.000000      0.190000  ...     61.000000     15.000000
## max     2020.000000      1.670000  ...  10665.000000     38.000000
##
## [8 rows x 11 columns]
```

```
data
```

```
##                                  Name Platform  ...  Rating Age
## 1058   Cabela's Big Game Hunter 2010      Wii  ...       T   9
## 1059        SOCOM 3: U.S. Navy SEALs      PS2  ...       M  13
## 1060               BioShock Infinite      PS3  ...       M   5
## 1061              Jampack Winter '99       PS  ...     NaN  19
## 1062          Call of Duty: Black Ops 3      PS3  ...     NaN   3
## ...                               ...      ...  ...     ...  ..
## 16714  Samurai Warriors: Sanada Maru      PS3  ...     NaN   2
## 16715                LMA Manager 2007     X360  ...     NaN  12
```

```
## 16716        Haitaka no Psychedelica      PSV  ...     NaN   2
## 16717              Spirits & Spells       GBA  ...     NaN  15
## 16718            Winning Post 8 2016      PSV  ...     NaN   2
##
## [15401 rows x 17 columns]
```

There are nearly half of the games which do not have scores. In ideal cases, you would like to drop these colunmns. But dropping over 8000+ entries is not possible in our case as it will heavily affect the models. Therefore, I am going to build 2 models: a basic one and an advanced model. In a basic model I will drop games without a score (critic or user) and train it on the remaining data. I will also do minimum feature engineering or feature selection.

After I am finished with the basic model, I am going to come back to the full dataset and try to impute missing values and create new features.

Basic Model

```
# Making a new column which shows if the game is scored or not. (User score and Critic Score)

data["Has_Score"] = data["User_Score"].notnull() & data["Critic_Score"].notnull()
rmvd_global["Has_Score"] = rmvd_global["User_Score"].notnull() & rmvd_global["Critic_Score"].notnull()
```

For my basic model I am going to drop games that don't have a user score, critic score or rating. I will also remove outliers in User_Count column. Only 5.5k games, ~1/3 of all games in a dataset remaining after doing the above steps.

```
scored = data.dropna(subset=["User_Score", "Critic_Score", "Rating"])
scored, rmvd_user_count = rm_outliers(scored, ["User_Count"])
scored.describe()
```

```
##               Year           NA  ...   User_Count          Age
## count  5534.000000  5534.000000  ...  5534.000000  5534.000000
## mean   2007.055837     0.205403  ...    37.459523    10.944163
## std       4.010373     0.225580  ...    44.572477     4.010373
## min    1985.000000     0.000000  ...     4.000000     2.000000
## 25%    2004.000000     0.060000  ...     9.000000     8.000000
## 50%    2007.000000     0.130000  ...    20.000000    11.000000
## 75%    2010.000000     0.280000  ...    45.000000    14.000000
## max    2016.000000     1.670000  ...   233.000000    33.000000
##
## [8 rows x 11 columns]
```

```
scored["Platform"].unique(), scored["Genre"].unique(), scored["Rating"].unique()
# 17 unique platfoms, 12 unique genres and 5 ratings are present in the given data.
```

```
## (array(['PS2', 'GBA', 'X360', 'PS3', 'PC', 'Wii', 'PSP', 'PS', 'XB', 'GC',
##        'DS', 'XOne', '3DS', 'DC', 'PS4', 'WiiU', 'PSV'], dtype=object), array(['Shooter', 'Action',
##        'Sports', 'Fighting', 'Platform', 'Misc', 'Strategy', 'Puzzle',
##        'Adventure'], dtype=object), array(['M', 'E', 'T', 'E10+', 'RP'], dtype=object))
```

There are 17 unique platfoms, 12 unique genres and 5 ratings in the remaining data. In the advanced model I will try grouping platforms to reduce amount, but for now I will just one-hot encode them.

Features will consist of numeric columns (except for sales in regions and year - using age instead) and one-hot encoded categorical columns (platform, genre, rating).

```
import category_encoders as ce

# Numeric columns
numeric_subset = scored.select_dtypes("number").drop(columns=["NA", "EU", "JP", "Other", "Year"])

# Categorical column
categorical_subset = scored[["Platform", "Genre", "Rating"]]

# One hot encoding
encoder = ce.one_hot.OneHotEncoder()
categorical_subset = encoder.fit_transform(categorical_subset)

# Column binding to the previos numeric dataset
```

```
## /Users/kenmai/Library/r-miniconda/envs/r-reticulate/lib/python3.9/site-packages/category_encoders/uti
##   elif pd.api.types.is_categorical(cols):
```

```
features = pd.concat([numeric_subset, categorical_subset], axis = 1)

# Find correlations with the score
correlations = features.corr()["Global"].dropna().sort_values()
```

Let's look at the highest and lowest correlations with the global sales column.

```
correlations.head()

# Platform 5 = PC
# Genre 10 = Strategy
# Genre 12 = Adventure
# Platform 17 = PSV
# Platform 15 = PS4
```

```
## Platform_5    -0.186725
## Genre_10      -0.094686
## Genre_12      -0.084227
## Platform_17   -0.069683
## Platform_15   -0.062370
## Name: Global, dtype: float64
```

```
correlations.tail()
```

```
## User_Score      0.155470
## User_Count      0.252651
## Critic_Score    0.281545
## Critic_Count    0.292327
## Global          1.000000
## Name: Global, dtype: float64
```

Splitting data into training set (80%) and test set (20%)

```
from sklearn.model_selection import train_test_split

X = features.drop(columns="Global")
Y = pd.Series(features["Global"])
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,test_size=0.2,random_state=42)

print(X_train.shape)
```

```
## (4427, 39)
```

In the next 2 cells I have: -

Defining function for mean absolute error Defining function for fitting the model

```
def mae(y_true, y_pred):
    return np.average(abs(y_true - y_pred))
```

```
def fit_and_evaluate(model):

    # Train the model
    model.fit(X_train, Y_train)

    # Make predictions and evalute
    model_pred = model.predict(X_test)
    model_mae = mae(Y_test, model_pred)

    # Return the performance metric
    return model_mae
```

I will compare several simple models with different types of regression, and then focus on the best one for hyperparameter tuning.

```
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.neural_network import MLPRegressor
```

```
baseline_guess = np.median(X_train)
basic_baseline_mae = mae(X_test, baseline_guess)
print("Baseline guess for global sales is: {:.02f}".format(baseline_guess))
```

```
## Baseline guess for global sales is: 0.00
```

```
print("Baseline Performance on the test set: MAE = {:.04f}".format(basic_baseline_mae))
```

```
## Baseline Performance on the test set: MAE = 3.9537
```

```
# Linear Regression
lr = LinearRegression()
lr_mae = fit_and_evaluate(lr)
print("Linear Regression Performance on the test set: MAE = {:.04f}".format(lr_mae))
```

```
## Linear Regression Performance on the test set: MAE = 0.2361
```

```
svm = SVR(C = 1000, gamma=0.1)
svm_mae = fit_and_evaluate(svm)

print("Support Vector Machine Regression Performance on the test set: MAE = {:.04f}".format(svm_mae))
```

```
## Support Vector Machine Regression Performance on the test set: MAE = 0.2859
```

```
random_forest = RandomForestRegressor(random_state=60)
random_forest_mae = fit_and_evaluate(random_forest)
print("Random Forest Regression Performance on the test set: MAE = {:.04f}".format(random_forest_mae))
```

```
## Random Forest Regression Performance on the test set: MAE = 0.2216
```

```
gradient_boosting = GradientBoostingRegressor(random_state=60)
gradient_boosting_mae = fit_and_evaluate(gradient_boosting)
print("Gradient Boosting Regression Performance on the test set: MAE = {:.04f}".format(gradient_boostin
```

```
## Gradient Boosting Regression Performance on the test set: MAE = 0.2197
```

```
knn = KNeighborsRegressor(n_neighbors=10)
knn_mae = fit_and_evaluate(knn)

print("K-Nearest Neighbors Regression Performance on the test set: MAE = {:.04f}".format(knn_mae))
```

```
## K-Nearest Neighbors Regression Performance on the test set: MAE = 0.2557
```

```
ridge = Ridge(alpha=10)
ridge_mae = fit_and_evaluate(ridge)

print("Ridge Regression Performance on the test set: MAE = {:.04f}".format(ridge_mae))
```

```
## Ridge Regression Performance on the test set: MAE = 0.2354
```

```
MLP = MLPRegressor(hidden_layer_sizes=(10, 10, 10), max_iter=1000)
MLP_mae = fit_and_evaluate(MLP)
print("MLP Regression Performance on the test set: MAE = {:.04f}".format(MLP_mae))
```

```
## MLP Regression Performance on the test set: MAE = 0.2239
```

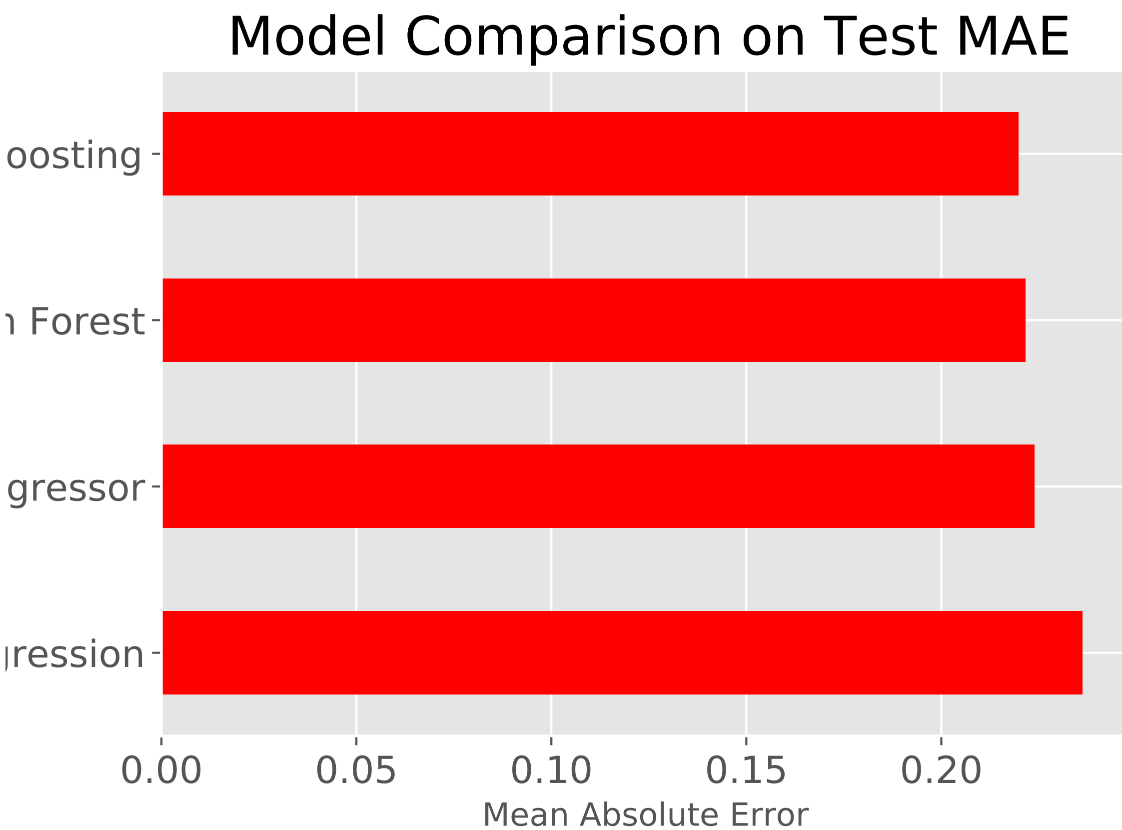
```
lasso = Lasso()
lasso_mae = fit_and_evaluate(lasso)

print("Lasso Regression Performance on the test set: MAE = {:.04f}".format(lasso_mae))
```

```
## Lasso Regression Performance on the test set: MAE = 0.2811
```

```
style.use('ggplot')
#model_comparison = pd.DataFrame({"model": ["Linear Regression", "Support Vector Machine","Random Fores
#                                             "K-Nearest Neighbors", "Ridge", "MLP Regressor", "Lasso"],
#                                  "mae": [lr_mae, svm_mae, random_forest_mae,
#                                          gradient_boosting_mae, knn_mae, ridge_mae, MLP_mae, lasso_mae

model_comparison = pd.DataFrame({"model": ["Linear Regression", "Random Forest", "Gradient Boosting","Ml
                                 "mae": [lr_mae, random_forest_mae, gradient_boosting_mae, MLP_mae]})

model_comparison.sort_values("mae", ascending=False).plot(x="model", y="mae", kind="barh",
                                                           color="red", legend=False)
plt.ylabel(""); plt.yticks(size=14); plt.xlabel("Mean Absolute Error"); plt.xticks(size=14)
```

```
## Text(0, 0.5, '')
## (array([0, 1, 2, 3]), [Text(0, 0, 'Linear Regression'), Text(0, 1, 'MLP Regressor'), Text(0, 2, 'Ranc
## Text(0.5, 0, 'Mean Absolute Error')
## (array([0.  , 0.05, 0.1 , 0.15, 0.2 , 0.25]), [Text(0, 0, ''), Text(0, 0, ''), Text(0, 0, ''), Text(0
```

```
plt.title("Model Comparison on Test MAE", size=20);

# Gradient Boosting is the best out of the 5 models chosen
```



Gradient boosting regressor seems to be the best model, I will focus on this one.

First I am going to use randomized search to find the best parameters, and then I will use grid search for optimizing n_estimators.

```
hyperparameter_grid = {"loss": ["ls", "lad", "huber"],
                       "max_depth": [2, 3, 5, 10, 15],
                       "min_samples_leaf": [1, 2, 4, 6, 8],
                       "min_samples_split": [2, 4, 6, 10],
                       "max_features": ["auto", "sqrt", "log2", None]}
```

```
from sklearn.model_selection import RandomizedSearchCV

basic_model = GradientBoostingRegressor(random_state = 42)
random_cv = RandomizedSearchCV(estimator=basic_model,
                               param_distributions=hyperparameter_grid,
                               cv=4, n_iter=20,
                               scoring="neg_mean_absolute_error",
                               n_jobs=-1, verbose=1,
                               return_train_score=True,
                               random_state=42)
```

```
random_cv.fit(X_train, Y_train)
```

```
## Fitting 4 folds for each of 20 candidates, totalling 80 fits
## RandomizedSearchCV(cv=4, estimator=GradientBoostingRegressor(random_state=42),
```

```
##                       n_iter=20, n_jobs=-1,
##                       param_distributions={'loss': ['ls', 'lad', 'huber'],
##                                            'max_depth': [2, 3, 5, 10, 15],
##                                            'max_features': ['auto', 'sqrt', 'log2',
##                                                             None],
##                                            'min_samples_leaf': [1, 2, 4, 6, 8],
##                                            'min_samples_split': [2, 4, 6, 10]},
##                       random_state=42, return_train_score=True,
##                       scoring='neg_mean_absolute_error', verbose=1)
```

Printing out 10 best estimators found by randomized search.

```
random_results = pd.DataFrame(random_cv.cv_results_).sort_values("mean_test_score", ascending=False)
random_results.head(10)[["mean_test_score", "param_loss",
                         "param_max_depth", "param_min_samples_leaf", "param_min_samples_split",
                         "param_max_features"]]
```

```
##     mean_test_score param_loss  ... param_min_samples_split param_max_features
## 0         -0.200985      huber  ...                       6               log2
## 17        -0.201954        lad  ...                       4               log2
## 7         -0.202492      huber  ...                       6               auto
## 16        -0.208397        lad  ...                      10               log2
## 15        -0.208956         ls  ...                       6               auto
## 8         -0.210707        lad  ...                      10               auto
## 19        -0.212971        lad  ...                      10               sqrt
## 3         -0.213049        lad  ...                      10               sqrt
## 1         -0.214555      huber  ...                       4               None
## 2         -0.216843         ls  ...                       4               sqrt
##
## [10 rows x 6 columns]
```

```
random_cv.best_estimator_
```

```
## GradientBoostingRegressor(loss='huber', max_depth=15, max_features='log2',
##                           min_samples_leaf=8, min_samples_split=6,
##                           random_state=42)
```

Using grid search to find optimal value of the n_estimators parameter.

```
# Using grid search to find optimal value of the n_estimators parameter.
from sklearn.model_selection import GridSearchCV
trees_grid = {"n_estimators": [50, 100, 150, 200, 250, 300]}

basic_model = random_cv.best_estimator_
grid_search = GridSearchCV(estimator=basic_model, param_grid=trees_grid, cv=4,
                           scoring="neg_mean_absolute_error", verbose=1,
                           n_jobs=-1, return_train_score=True)
```

```
grid_search.fit(X_train, Y_train)
```

```
## Fitting 4 folds for each of 6 candidates, totalling 24 fits
```

```
## GridSearchCV(cv=4,
##              estimator=GradientBoostingRegressor(loss='huber', max_depth=15,
##                                                  max_features='log2',
##                                                  min_samples_leaf=8,
##                                                  min_samples_split=6,
##                                                  random_state=42),
##              n_jobs=-1,
##              param_grid={'n_estimators': [50, 100, 150, 200, 250, 300]},
##              return_train_score=True, scoring='neg_mean_absolute_error',
##              verbose=1)
```

```
grid_search.best_estimator_
```

```
## GradientBoostingRegressor(loss='huber', max_depth=15, max_features='log2',
##                           min_samples_leaf=8, min_samples_split=6,
##                           n_estimators=50, random_state=42)
```

```
grid_search.fit(X_train, Y_train)
```

```
## Fitting 4 folds for each of 6 candidates, totalling 24 fits
## GridSearchCV(cv=4,
##              estimator=GradientBoostingRegressor(loss='huber', max_depth=15,
##                                                  max_features='log2',
##                                                  min_samples_leaf=8,
##                                                  min_samples_split=6,
##                                                  random_state=42),
##              n_jobs=-1,
##              param_grid={'n_estimators': [50, 100, 150, 200, 250, 300]},
##              return_train_score=True, scoring='neg_mean_absolute_error',
##              verbose=1)
```

```
results = pd.DataFrame(grid_search.cv_results_)

plt.plot(results["param_n_estimators"], -1 * results["mean_test_score"], label = "Testing Error")
plt.plot(results["param_n_estimators"], -1 * results["mean_train_score"], label = "Training Error")
plt.xlabel("Number of Trees"); plt.ylabel("Mean Abosolute Error"); plt.legend();
plt.title("Performance vs Number of Trees");
```

The graph shows that the model is overfitting. Training error keeps decreasing, while test error stays almost the same. It means that the model learns training examples very well, but cannot generalize on new, unknown data. This is not a very good model and try to battle overfitting in the advanced model using imputing, feature selection and feature engineering.

Let's lock the final model and see how it performs on test data.

```
basic_final_model = grid_search.best_estimator_
basic_final_pred = basic_final_model.predict(X_test)
basic_final_mae = mae(Y_test, basic_final_pred)
print("Final model performance on the test set: MAE = {:.04f}.".format(basic_final_mae))
```

```
## Final model performance on the test set: MAE = 0.2086.
```

MAE dropped, but by a very small margin. Looks like hyperparameter tuning didn't really improve the model. I hope advanced model will have a better performance. To finish with the basic model I am going to draw 2 graphs. First one is comparison of densities of train values, test values and predictions.

```
# Density plots for predictions ,test, train

sns.kdeplot(basic_final_pred, label = "Predictions")
sns.kdeplot(Y_test, label = "Test")
```

```
## <AxesSubplot:title={'center':'Performance vs Number of Trees'}, xlabel='Number of Trees', ylabel='Mea
```

```
sns.kdeplot(Y_train, label = "Train")
```

```
## <AxesSubplot:title={'center':'Performance vs Number of Trees'}, xlabel='Number of Trees', ylabel='Mea
```

```
plt.xlabel("Global Sales"); plt.ylabel("Density");
plt.title("Test, Train Values and Predictions");
```

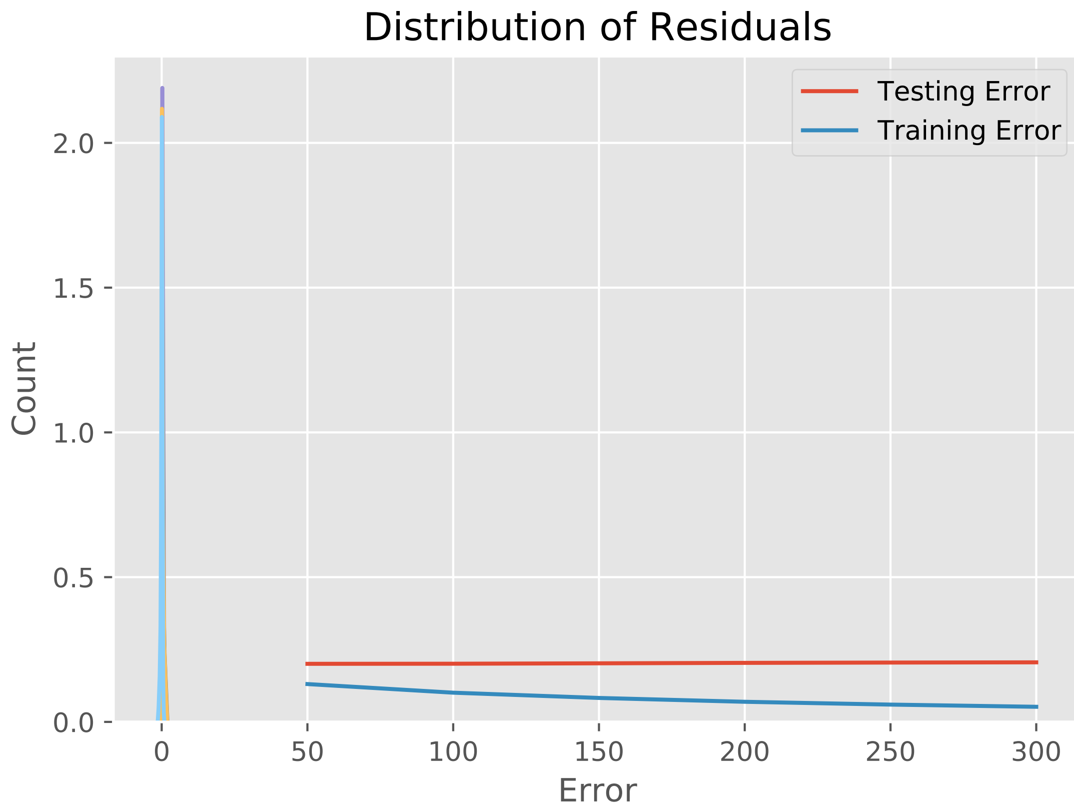
Predictions density is moved a little to the right, comparing to densities of initial values. The tail is also different. This might help tuning the model in the future.

Second graph is a histogram of residuals - differences between real values and predictions.

```
# Residuals plot

basic_residuals = basic_final_pred - Y_test

sns.kdeplot(basic_residuals, color = "lightskyblue")
```

```
## <AxesSubplot:title={'center':'Test, Train Values and Predictions'}, xlabel='Global Sales', ylabel='De
```

```
plt.xlabel("Error"); plt.ylabel("Count")
plt.title("Distribution of Residuals")
```



Advanced Model

```
#Number of unique platforms present
data["Platform"].unique()
```

```
## array(['Wii', 'PS2', 'PS3', 'PS', 'N64', 'GBA', 'DS', 'GC', 'X360', 'GB',
##        'PC', '3DS', 'PSP', 'XB', 'NES', 'PS4', 'GEN', '2600', 'SNES',
##        'XOne', 'WiiU', 'PSV', 'SCD', 'DC', 'SAT', 'WS', 'NG', 'TG16',
##        '3DO', 'GG', 'PCFX'], dtype=object)
```

There are too many different platforms and most of them represent a very small percent of games. I am going to group platforms to reduce the number of features.

```
#Grouping platforms together
platforms = {"Playstation" : ["PS", "PS2", "PS3", "PS4"],
             "Xbox" : ["XB", "X360", "XOne"],
             "PC" : ["PC"],
             "Nintendo" : ["Wii", "WiiU"],
             "Portable" : ["GB", "GBA", "GC", "DS", "3DS", "PSP", "PSV"]}
```

Below are the functions that I am going to use to plot the data and get inferences as well help to group the platforms as required.

```
def visual_chart(column, palette="Set2"):
    values = column.value_counts().values
    labels = column.value_counts().index
    plt.pie(values, colors=sns.color_palette(palette),
            labels=labels, autopct="%1.1f%%",
            startangle=90, pctdistance=0.85)

    #draw circle
    centre_circle = plt.Circle((0,0), 0.70, fc="white")
    fig = plt.gcf()
    fig.gca().add_artist(centre_circle)
```

```
def get_group_label(x, groups=None):
    if groups is None:
        return "Other"
    else:
        for key, val in groups.items():
            if x in val:
                return key
        return "Other"
```

```
data["Grouped_Platform"] = data["Platform"].apply(lambda x: get_group_label(x, groups=platforms))
visual_chart(data["Grouped_Platform"])
plt.title("Groups of platforms")
plt.axis("equal");
```

Looks much better.

Now I want to check the same thing for genres.

```
visual_chart(data["Genre"], palette="muted")
plt.title("Genres")
```

```
## Text(0.5, 1.0, 'Genres')
```

```
plt.axis("equal")
```

```
## (-1.1161806247690709, 1.111197844931594, -1.1055472052999136, 1.1002641572435603)
```

The distribution seems ok, even though there is a significant number of different genres.

```
#Grouping the platforms for the entries whose score is given
scored["Grouped_Platform"] = scored["Platform"].apply(lambda x: get_group_label(x, platforms))
visual_chart(scored["Grouped_Platform"])
plt.title("Groups of platforms for games with score")
plt.axis("equal");
```

Almost all games that have scores are for "big" platfroms: PC, PS, Xbox or portable. But there are few from the "Other" group. Below are the results what the "Other" platform represents. (DC - Dreamcast)

```
scored[scored["Grouped_Platform"]=="Other"]
```

```
##                            Name Platform  ...  Has_Score Grouped_Platform
## 1712                      Shenmue       DC  ...       True            Other
## 1877                      NFL 2K1       DC  ...       True            Other
## 3815                       Seaman       DC  ...       True            Other
## 5350                   SoulCalibur       DC  ...       True            Other
## 7231                Capcom vs. SNK       DC  ...       True            Other
## 7521           Phantasy Star Online       DC  ...       True            Other
## 7643                    Grandia II       DC  ...       True            Other
## 7978   Phantasy Star Online Ver. 2       DC  ...       True            Other
## 8905                    Shenmue II       DC  ...       True            Other
## 9559                       Sega GT       DC  ...       True            Other
## 10999             Skies of Arcadia       DC  ...       True            Other
## 12096                 Crazy Taxi 2       DC  ...       True            Other
## 13110        The Typing of the Dead       DC  ...       True            Other
##
## [13 rows x 19 columns]
```

Next I want to create some new features: weighted score and my own developer rating. First, I find percent of all games created by each developer, then calculate cumulative percent starting with devs with the least number of games. Finally, I divide them into 5 groups (20% each). Higher rank means more games developed.

Higher top percentage means more games developed.

```
# One weighted score value including all scores and counts field.
scored["Weighted_Score"] = (scored["User_Score"] * 10 * scored["User_Count"] +
                            scored["Critic_Score"] * scored["Critic_Count"]) / (scored["User_Count"] + s

# Dataframe having developers arranged based on their frequency
devs = pd.DataFrame({"dev": scored["Developer"].value_counts().index,
                     "count": scored["Developer"].value_counts().values})

# Mean scoring datafram based on the weighted score
m_score = pd.DataFrame({"dev": scored.groupby("Developer")["Weighted_Score"].mean().index,
                        "mean_score": scored.groupby("Developer")["Weighted_Score"].mean().values})

# Creating merging the mean_score and developer dataframes and then sorting the resultant into ascendin
devs = pd.merge(devs, m_score, on="dev")
devs = devs.sort_values(by="count", ascending=True)

# Percentage of all games created by each developer and storing it in form of cumulative fashion
devs["percent"] = devs["count"] / devs["count"].sum()
devs["top%"] = devs["percent"].cumsum() * 100

# Dividing them into 10 groups
n_groups = 10
devs["top_group"] = (devs["top%"] * n_groups) // 100 + 1
devs["top_group"].iloc[-1] = n_groups
```

```
## /Users/kenmai/Library/r-miniconda/envs/r-reticulate/lib/python3.9/site-packages/pandas/core/indexing
## A value is trying to be set on a copy of a slice from a DataFrame
##
```

```
## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexin
##   self._setitem_single_block(indexer, value, name)
```

```
devs
```

```
##                          dev  count  ...        top%  top_group
## 1179  SCE/WWS, SCE Japan Studio      1  ...    0.018070        1.0
## 842            Babylon Software      1  ...    0.036140        1.0
## 843                  Felistella      1  ...    0.054210        1.0
## 844          Direct Action Games      1  ...    0.072280        1.0
## 845               Lab Rats Games      1  ...    0.090351        1.0
## ...                         ...    ...  ...         ...        ...
## 4                        Konami     78  ...   92.410553       10.0
## 3                       Ubisoft     86  ...   93.964583       10.0
## 2                        Capcom     92  ...   95.627033       10.0
## 1                     EA Sports    116  ...   97.723166       10.0
## 0                     EA Canada    126  ...  100.000000       10.0
##
## [1180 rows x 6 columns]
```

Before creating and fitting a model I have to fill in missing values. I am filling scores and counts with zeros, because there were no real zero scores or counts in the dataset, so it will indicate absence of scores.

```
data["Critic_Score"].fillna(0.0, inplace=True)
data["Critic_Count"].fillna(0.0, inplace=True)
data["User_Score"].fillna(0.0, inplace=True)
data["User_Count"].fillna(0.0, inplace=True)
data = data.join(devs.set_index("dev")["top_group"], on="Developer")
data = data.rename(columns={"top_group": "Developer_Rank"})
data["Developer_Rank"].fillna(0.0, inplace=True)
data["Rating"].fillna("None", inplace=True)
```

Removing outliers in User_Count columns.

```
tmp, rmvd_tmp = rm_outliers(data[data["User_Count"] != 0], ["User_Count"])
data.drop(rmvd_tmp.index, axis=0, inplace=True)
```

Creating Weighted_Score column (earlier I did it for “scored” dataframe).

```
data["Weighted_Score"] = (data["User_Score"] * 10 * data["User_Count"] +
                            data["Critic_Score"] * data["Critic_Count"]) / (data["User_Count"] + data["C
data["Weighted_Score"].fillna(0.0, inplace=True)
```

```
data.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## Int64Index: 14743 entries, 1058 to 16718
## Data columns (total 21 columns):
##  #   Column            Non-Null Count  Dtype
## ---  ------            --------------  -----
##  0   Name              14743 non-null  object
```

```
## 1   Platform          14743 non-null  object
## 2   Year              14743 non-null  int64
## 3   Genre             14743 non-null  object
## 4   Publisher         14712 non-null  object
## 5   NA                14743 non-null  float64
## 6   EU                14743 non-null  float64
## 7   JP                14743 non-null  float64
## 8   Other             14743 non-null  float64
## 9   Global            14743 non-null  float64
## 10  Critic_Score      14743 non-null  float64
## 11  Critic_Count      14743 non-null  float64
## 12  User_Score        14743 non-null  float64
## 13  User_Count        14743 non-null  float64
## 14  Developer         8519 non-null   object
## 15  Rating            14743 non-null  object
## 16  Age               14743 non-null  int64
## 17  Has_Score         14743 non-null  bool
## 18  Grouped_Platform  14743 non-null  object
## 19  Developer_Rank    14743 non-null  float64
## 20  Weighted_Score    14743 non-null  float64
## dtypes: bool(1), float64(11), int64(2), object(7)
## memory usage: 2.4+ MB
```

Now I will do the same things as I did in the basic model, except for using Ordinal encoding for categorical values instead of OneHot.

```python
# Select the numeric columns
numeric_subset = data.select_dtypes("number").drop(columns=["NA", "EU", "JP", "Other", "Year"])

# Select the categorical columns
categorical_subset = data[["Grouped_Platform", "Genre", "Rating"]]

mapping = []
for cat in categorical_subset.columns:
    tmp = scored.groupby(cat).median()["Weighted_Score"]
    mapping.append({"col": cat, "mapping": [x for x in np.argsort(tmp).items()]})

encoder = ce.ordinal.OrdinalEncoder()
categorical_subset = encoder.fit_transform(categorical_subset, mapping=mapping)

# Join the two dataframes using concat. Axis = 1 -> Column bind
features = pd.concat([numeric_subset, categorical_subset], axis = 1)

# Find correlations with the score
correlations = features.corr()["Global"].dropna().sort_values()
```

```python
features
```

```
##        Global  Critic_Score  Critic_Count  ...  Grouped_Platform  Genre  Rating
## 1058     1.69           0.0           0.0  ...                 1      1       1
## 1059     1.69          82.0          59.0  ...                 2      2       2
## 1061     1.69           0.0           0.0  ...                 2      3       3
## 1062     1.69           0.0           0.0  ...                 2      2       3
```

```
## 1063     1.69           0.0           0.0  ...                3      4       3
## ...       ...           ...           ...  ...              ...    ...     ...
## 16714    0.01           0.0           0.0  ...                2      5       3
## 16715    0.01           0.0           0.0  ...                5      1       3
## 16716    0.01           0.0           0.0  ...                4     11       3
## 16717    0.01           0.0           0.0  ...                4     10       3
## 16718    0.01           0.0           0.0  ...                4      7       3
##
## [14743 rows x 11 columns]
```

Dividing the final data into training and testing. After that I applyied the gradient boosting algorithm and then fitting the respective hyperparameters by using randomized search to do so.

```
target = pd.Series(features["Global"])
features = features.drop(columns="Global")
features_train, features_test, target_train, target_test = train_test_split(features, target, test_size=
```

```
model = GradientBoostingRegressor(random_state = 42)

random_cv = RandomizedSearchCV(estimator=model,
                               param_distributions=hyperparameter_grid,
                               cv=4, n_iter=20,
                               scoring="neg_mean_absolute_error",
                               n_jobs=-1, verbose=1,
                               return_train_score=True,
                               random_state=42)
random_cv.fit(features_train, target_train);
```

```
## Fitting 4 folds for each of 20 candidates, totalling 80 fits
```

```
trees_grid = {"n_estimators": [50, 100, 150, 200, 250, 300]}

model = random_cv.best_estimator_
grid_search = GridSearchCV(estimator=model, param_grid=trees_grid, cv=4,
                           scoring="neg_mean_absolute_error", verbose=1,
                           n_jobs=-1, return_train_score=True)
grid_search.fit(features_train, target_train);
```

```
## Fitting 4 folds for each of 6 candidates, totalling 24 fits
```

```
# Getting the final model error
final_model = grid_search.best_estimator_
final_pred = final_model.predict(features_test)
final_mae = mae(target_test, final_pred)
print("Final model performance on the test set: MAE = {:.04f}.".format(final_mae))
```

```
## Final model performance on the test set: MAE = 0.1752.
```

“Advanced” model gives better results (lower error on test set) which is a good achievement. There is definitely room for improvement. And to finish with the project, a nice group of plots summarizing the results.

```python
# Final Comparison Graph

plt.figure(figsize=(20, 16))
plt.title("Video Games - Predicting Global Sales", size=30, weight="bold");

ax=plt.subplot(2, 2, 1)
sns.kdeplot(final_pred, color="limegreen", label="Advanced Model")
sns.kdeplot(basic_final_pred, color="indianred", label="Basic Model")
```

```
## <AxesSubplot:ylabel='Density'>
```

```python
sns.kdeplot(target_test, color="royalblue", label="Test")
```

```
## <AxesSubplot:xlabel='Global', ylabel='Density'>
```

```python
plt.xlabel("Global Sales, $M", size=20)
plt.ylabel("Density", size=20)
plt.title("Distribution of Target Values", size=24)

residuals = final_pred - target_test
ax =plt.subplot(2, 2, 2)
sns.kdeplot(residuals, color = "limegreen", label="Advanced Model")
sns.kdeplot(basic_residuals, color="indianred", label="Basic Model")
```

```
## <AxesSubplot:xlabel='Global', ylabel='Density'>
```

```python
plt.xlabel("Residuals, $M", size=20)
plt.ylabel("Density", size=20);
plt.title("Distribution of Errors", size=24)

feature_importance = final_model.feature_importances_
feature_names = features.columns.tolist()
feature_importance = 100.0 * (feature_importance / feature_importance.max())
sorted_idx = np.argsort(feature_importance)
pos = np.arange(sorted_idx.shape[0]) + .5
ax =plt.subplot(2, 2, 3)
plt.barh(pos, feature_importance[sorted_idx], align='center', color="goldenrod")
```

```
## <BarContainer object of 10 artists>
```

```python
plt.yticks(pos, [feature_names[x] for x in sorted_idx], size=16)
```

```
## ([<matplotlib.axis.YTick object at 0x13bfeeaf0>, <matplotlib.axis.YTick object at 0x13c0a38b0>, <mat
```

```python
plt.xlabel('Relative Importance', size=20)
plt.title('Variable Importance', size=24)

model_comparison = pd.DataFrame({"model": ["Baseline", "Basic", "Advanced"],
                                 "mae": [basic_baseline_mae, basic_final_mae, final_mae],
                                 "color": ["royalblue", "indianred", "limegreen"]})
model_comparison.sort_values("mae", ascending=False)
```

```
##       model       mae      color
## 0  Baseline  3.953712  royalblue
## 1     Basic  0.208577  indianred
## 2  Advanced  0.175241  limegreen
```

```
pos = np.arange(3) + .5
ax =plt.subplot(2, 2, 4)
plt.barh(pos, model_comparison["mae"], align="center", color=model_comparison["color"])
```

```
## <BarContainer object of 3 artists>
```

```
plt.yticks(pos, model_comparison["model"], size=16); plt.xlabel("Mean Absolute Error", size=20);
plt.title("Test MAE", size=24)
```