# SC1015
# Mini project


FCSI Group 4
Kalaiselvan Shanmugapriyan (U2323515D)
Manikandan Yuvana (U2322251E)
Rajadharshini Nedumaran (U2322490L)

# TABLE OF CONTENTS

# Video Games Sales & Rating

Source from: Kaggle

# Description of Dataset

| | |
|---|---|
| Categorical | 1. Name<br>2. Platform<br>3. Year_of_Release<br>4. Genre<br>5. Publisher<br>6. Developer<br>7. Rating |
| Numeric | 1. NA_Sales, EU_Sales, JP_Sales & Other_Sales<br>2. Global_Sales<br>3. Critic_Score & User_score<br>4. Critic_Count & User_Count |

# Description of Dataset

| | index | Name | Platform | Year_of_Release | Genre | Publisher | NA_Sales | EU_Sales | JP_Sales | Other_Sales | Global_Sales | Critic_Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | Wii Sports | Wii | 2006.0 | Sports | Nintendo | 41.36 | 28.96 | 3.77 | 8.45 | 82.53 | 76.0 |
| **1** | 1 | Super Mario Bros. | NES | 1985.0 | Platform | Nintendo | 29.08 | 3.58 | 6.81 | 0.77 | 40.24 | NaN |
| **2** | 2 | Mario Kart Wii | Wii | 2008.0 | Racing | Nintendo | 15.68 | 12.76 | 3.79 | 3.29 | 35.52 | 82.0 |
| **3** | 3 | Wii Sports Resort | Wii | 2009.0 | Sports | Nintendo | 15.61 | 10.93 | 3.28 | 2.95 | 32.77 | 80.0 |
| **4** | 4 | Pokemon Red/Pokemon Blue | GB | 1996.0 | Role-Playing | Nintendo | 11.27 | 8.89 | 10.22 | 1.00 | 31.37 | NaN |

raw data: (16928,17)

# Motivation

Predicting the **global sales** from various game-related variables such as domestic sales, ratings, genre, platform, user & critic scores and gain insights into **consumer preferences** and **market dynamics.**

# Our Models

Linear Regression

Lasso Regression

Radial Basis Function Network

# Data Cleaning

# Removing

```
main_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16928 entries, 0 to 16927
Data columns (total 17 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   index            16928 non-null  int64
 1   Name             16926 non-null  object
 2   Platform         16928 non-null  object
 3   Year_of_Release  16655 non-null  float64
 4   Genre            16926 non-null  object
 5   Publisher        16873 non-null  object
 6   NA_Sales         16928 non-null  float64
 7   EU_Sales         16928 non-null  float64
 8   JP_Sales         16928 non-null  float64
 9   Other_Sales      16928 non-null  float64
 10  Global_Sales     16928 non-null  float64
 11  Critic_Score     8260 non-null   float64
 12  Critic_Count     8260 non-null   float64
 13  User_Score       10159 non-null  object
 14  User_Count       7718 non-null   float64
 15  Developer        10240 non-null  object
 16  Rating           10092 non-null  object
dtypes: float64(9), int64(1), object(7)
memory usage: 2.2+ MB
```

Null Values

User_Score to numeric

Outliers

Dropped JP_Sales & User_Count

```
Number of outliers of index :
0

Number of outliers of NA_Sales :
686

Number of outliers of EU_Sales :
771

Number of outliers of JP_Sales :
1516

Number of outliers of Other_Sales :
799

Number of outliers of Critic_Score :
100

Number of outliers of Critic_Count :
176

Number of outliers of User_Score :
268

Number of outliers of User_Count :
1006
```

# Target Encoding

1. Regression models needs numeric inputs
2. Need to convert categorical inputs into numeric inputs
3. **Target encoding:** replaces the object input with a number that represents the mean target value for each category
4. Calculated: mean Global_Sale for that category
5. Target encoding for  categorical values Platform, Genre, Publisher, Developer, Rating

| | Platform | mean_Platform |
|---|---|---|
| **4653** | WiiU | 0.134103 |
| **4720** | PS2 | 0.152714 |
| **4751** | PS3 | 0.180270 |
| **4788** | GC | 0.141435 |
| **4813** | PSV | 0.144886 |
| **4875** | Wii | 0.160289 |
| **4877** | PS3 | 0.180270 |
| **4883** | 3DS | 0.167342 |
| **4905** | 3DS | 0.167342 |
| **4911** | Wii | 0.160289 |

# Cleaned Data

cleaned data:
(3186,20)

```
main_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3186 entries, 4653 to 16753
Data columns (total 20 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   index             3186 non-null   int64  
 1   Name              3186 non-null   object 
 2   Platform          3186 non-null   object 
 3   Year_of_Release   3186 non-null   float64
 4   Genre             3186 non-null   object 
 5   Publisher         3186 non-null   object 
 6   NA_Sales          3186 non-null   float64
 7   EU_Sales          3186 non-null   float64
 8   Other_Sales       3186 non-null   float64
 9   Global_Sales      3186 non-null   float64
 10  Critic_Score      3186 non-null   float64
 11  Critic_Count      3186 non-null   float64
 12  User_Score        3186 non-null   float64
 13  Developer         3186 non-null   object 
 14  Rating            3186 non-null   object 
 15  mean_Platform     3186 non-null   float64
 16  mean_Genre        3186 non-null   float64
 17  mean_Publisher    3186 non-null   float64
 18  mean_Developer    3186 non-null   float64
 19  mean_Rating       3186 non-null   float64
dtypes: float64(13), int64(1), object(6)
memory usage: 522.7+ KB
```

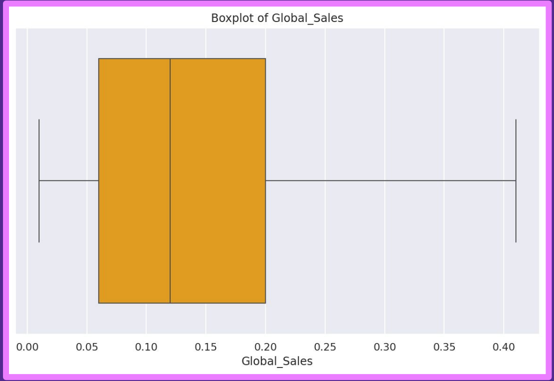
# EDA

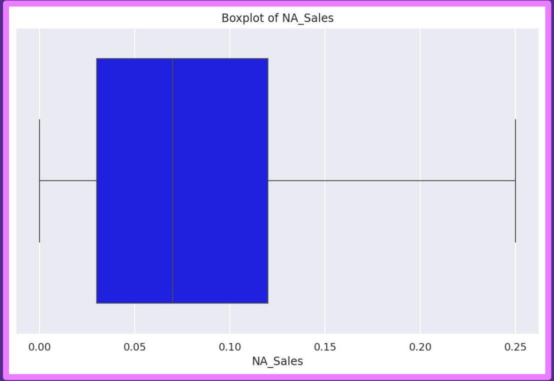
# Univariate (Numeric)

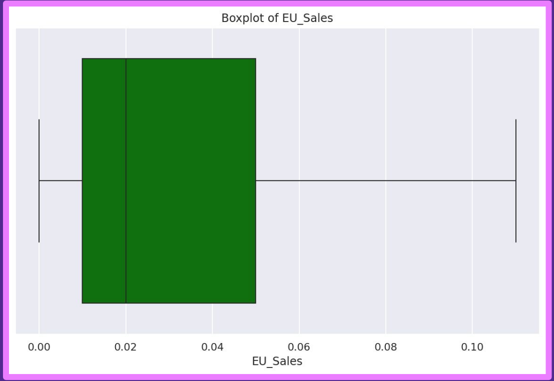
Boxplot

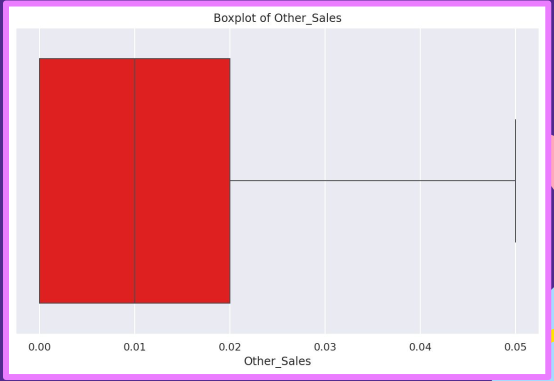
Histograms

Show median, quartiles, symmetry and spread of data. Spread of data similar for the all sales. Global_Sales highest mean followed by NA_Sales, EU_Sales and Other_Sales



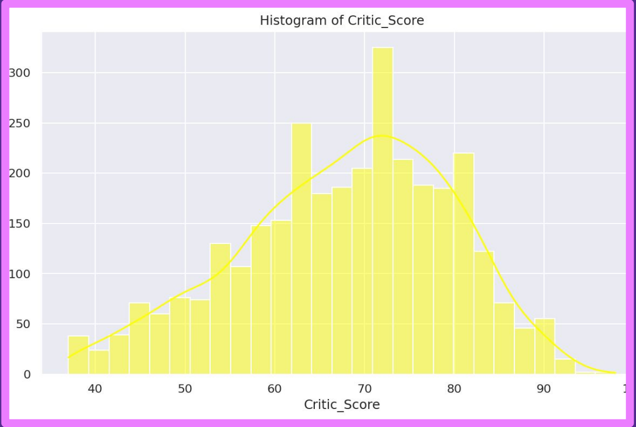



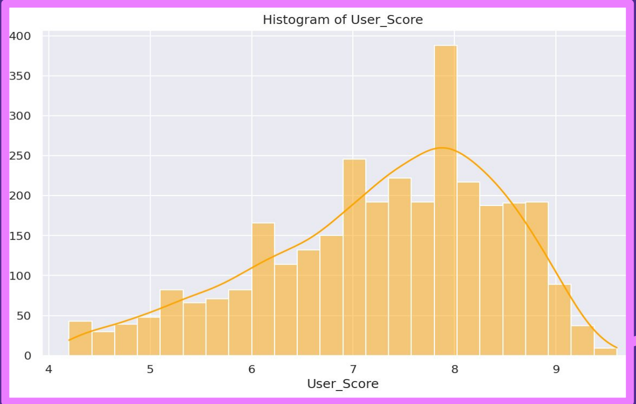



# Univariate (Numeric)

## Histograms

Show distribution shape, central tendency, spread of data and most importantly **frequency.** From histograms we can analyse that Critic_Score of 72 and User_Score of about 7.8 have highest frequency.



# Univariate (Categorical)

## Histograms

Grouped Year_of_Release as there were too many categories

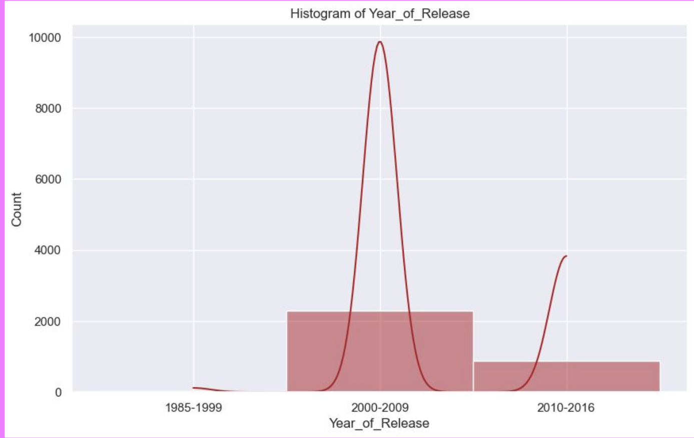
1. 1985-1999
2. 2000-2009
3. 2010-2016

```
# Sample DataFrame setup
# main_data = pd.DataFrame({
#     'Year_of_Release': [1980, 1987, 1995, 2001, 2005, 2010, 2014, 2017]
# })

# Define bins and labels for the categorization
bins = [1984, 1999, 2009, 2016, float('inf')]
labels = ['1985-1999', '2000-2009', '2010-2016', 'Post-2016']

# Categorize years using cut
main_data['Year_of_Release'] = pd.cut(main_data['Year_of_Release'], bins=bins, labels=labels, right=True)

# Check the result
main_data.head()
```

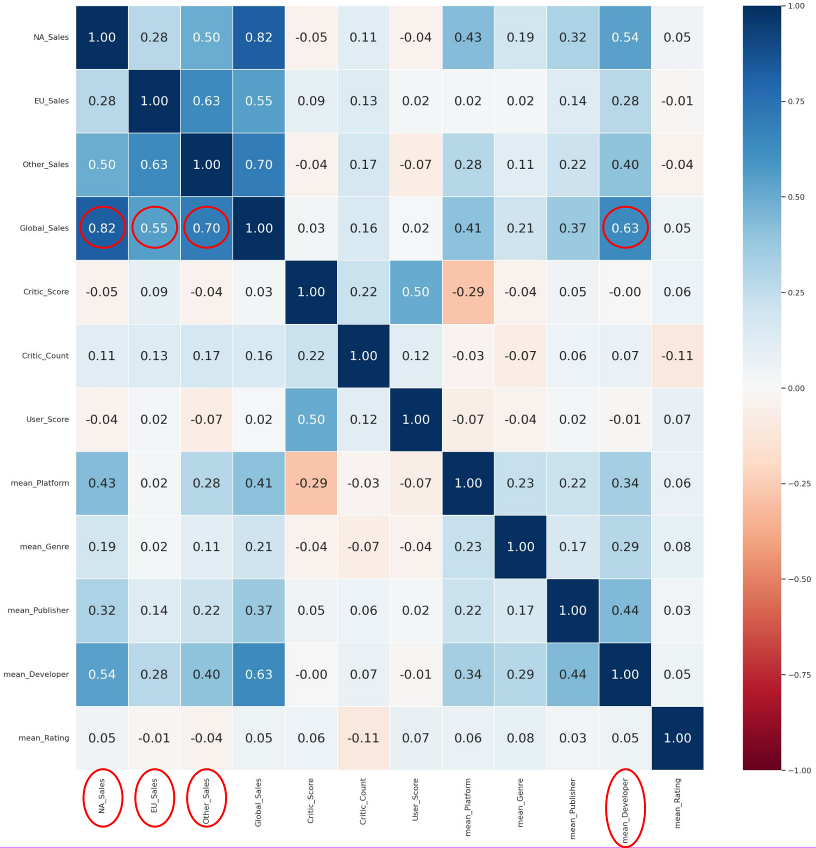


# Multivariate

## Correlation Matrix

Display the degree of correlation between multiple variables.

Columns with high correlation with Global_Sales are **NA_Sales (0.82)**, **Other_Sales (0.70)**, EU_Sales (0.55) and **mean_Developer (0.63)**



# Data Preparation

# Train and Test data

| | |
|---|---|
| Train Data | 1985 - 2009 |
| Test Data | 2010-2016 |

Long-term trends, seasonal variations, and shifts in patterns over time, and models can be used for time-series forecasting.

# Linear Regression

# What is Linear Regression?

1 **Definition** — Statistical method used for modeling the relationship between a dependent variable and one or more independent variables

2 **Function** — Fit a linear equation to the observed data points to best explain the relationship between the variables

3 **Equation** —

$$\underbrace{Y_i}_{\text{Dependent Variable}} = \overbrace{\beta_0}^{\text{Constant/Intercept}} + \underbrace{\beta_1}_{\text{Slope/Coefficient}} \overbrace{X_i}^{\text{Independent Variable}}$$

# Code breakdown

1. Data preparation
2. Model Fitting and Prediction
3. Model Evaluation (Finding intercept and coefficient)
4. Results Storage
5. Visualisation using regression line
6. DataFrame Creation

# Numerical Columns: Results

| | Numeric_Columns | Intercept | Coefficient | Explained Variance (R^2) | Mean Squared Error (MSE) | Root Mean Squared Error (RMSE) |
|---|---|---|---|---|---|---|
| 0 | NA_Sales | [0.03431150532383362] | [[1.1729032092117528]] | 0.544756 | 0.003901 | 0.062455 |
| 1 | EU_Sales | [0.0806625861972747] | [[1.8266635892450092]] | 0.223909 | 0.006650 | 0.081546 |
| 2 | Other_Sales | [0.0742045066073027] | [[6.243742322558052]] | 0.532308 | 0.004007 | 0.063303 |
| 3 | Critic_Score | [0.11566661332890484] | [[0.0002539577662829581]] | -0.001720 | 0.008583 | 0.092644 |
| 4 | Critic_Count | [0.11296655583315984] | [[0.0008813888936803135]] | 0.047500 | 0.008161 | 0.090340 |
| 5 | User_Score | [0.12918207984238958] | [[0.0005001121337913807]] | -0.001469 | 0.008581 | 0.092633 |

# Categorical Columns: Results

| | Categorical_Columns | Intercept | Coefficient | Explained Variance (R^2) | Mean Squared Error (MSE) | Root Mean Squared Error (RMSE) |
|---|---|---|---|---|---|---|
| 0 | mean_Platform | [-0.02293693431815408] | [[1.1472728071350073]] | 0.128325 | 0.007469 | 0.086422 |
| 1 | mean_Genre | [0.010144210707715984] | [[0.9237169141415654]] | 0.049086 | 0.008148 | 0.090264 |
| 2 | mean_Publisher | [-0.003168052280176137] | [[1.0114283738921628]] | 0.146923 | 0.007309 | 0.085495 |
| 3 | mean_Developer | [0.0003642005507176749] | [[0.9867659447434065]] | 0.431708 | 0.004869 | 0.069780 |
| 4 | mean_Rating | [-0.027304667139201017] | [[1.192478098091714]] | -0.000871 | 0.008576 | 0.092605 |

# Lasso Regression

# What is Lasso Regression?

1. **Definition** → Regularization technique that applies a penalty to prevent overfitting

2. **Function** → Enhance the accuracy of statistical linear regression models

3. **Equation** →

$$\text{minimize}\left(\text{RSS} + \lambda \sum_{j=1}^{p} |\beta_j|\right)$$

# Process of coding Lasso

Numerical Columns

Categorical Columns

All columns

# Code breakdown

1. Data preparation
2. Lasso Regression Model without Hyperparameter Tuning
3. Hyperparameter Tuning using GridSearchCV (alpha: 0.01)
    a. param_grid = { 'alpha': [0.01, 0.1, 1, 10, 100, 1000] }
4. Lasso Regression Model with Hyperparameter Tuning
5. Visualisation using KDE plot

# Numerical Columns: Results

**Metrics for Lasso model without hyperparameter tuning:**

- MAE: 0.07659789363324908
- MSE: 0.00858404990760388
- R^2 Score: -0.0018429077763817414

**Best hyperparameters found by GridSearchCV:** alpha = 0.01

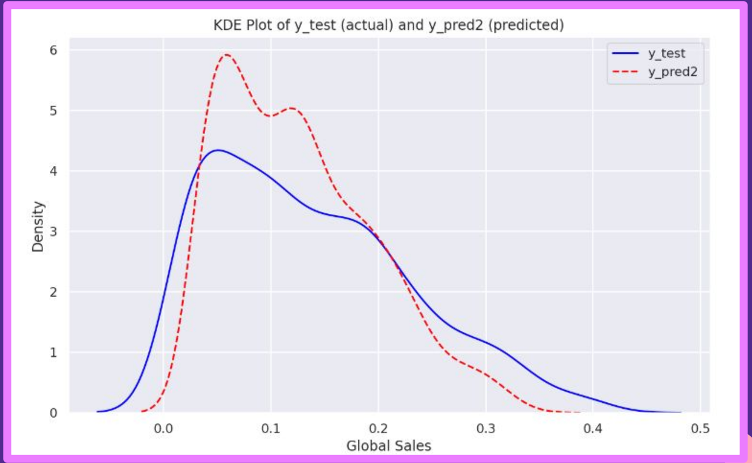
**Metrics for Lasso model with hyperparameter tuning:**

- MAE: 0.029376156814276903
- MSE: 0.0023583904186542043
- R^2 Score: 0.7247526819941321

**Intercept of Lasso model with best hyperparameters:**
0.13290056448111157

**Coefficients of Lasso model with best hyperparameters:**

0.05353402 0.01565545, 0.01354986, 0, 0, 0



# Categorical Columns: Results

**Metrics for Lasso model without hyperparameter tuning:**

- MAE: 0.07659789363324908
- MSE: 0.00858404990760388
- R^2 Score: -0.0018429077763817414

**Best hyperparameters found by GridSearchCV:** alpha = 0.01

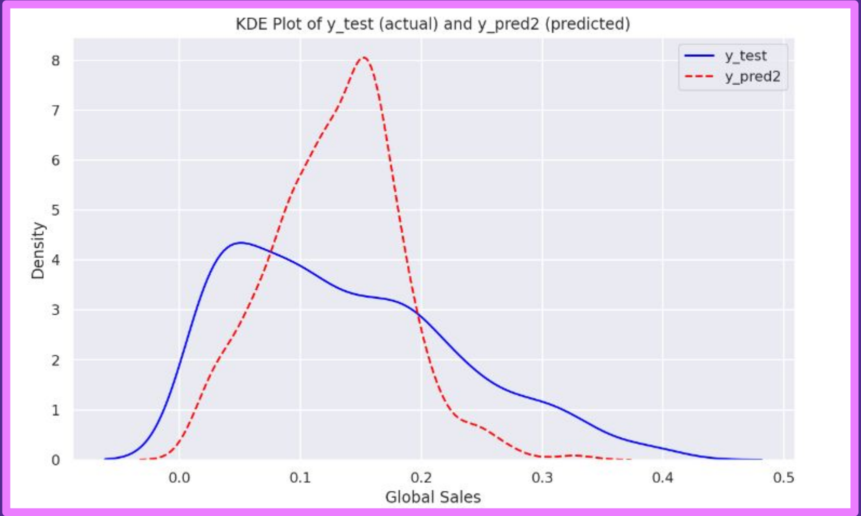
**Metrics for Lasso model with hyperparameter tuning:**

- MAE: 0.0519497308344659
- MSE: 0.0046613346947428445
- R^2 Score: 0.4559764732304953

**Intercept of Lasso model with best hyperparameters:**
0.1329005644811116

**Coefficients of Lasso model with best hyperparameters:**

0.01304357, 0, 0.00246752, 0.03957949, 0



# Numerical and Categorical Columns: Results

**Metrics for Lasso model without hyperparameter tuning:**

- MAE: 0.07659789363324908
- MSE: 0.00858404990760388
- R^2 Score: -0.0018429077763817414

**Best hyperparameters found by GridSearchCV:** alpha = 0.01

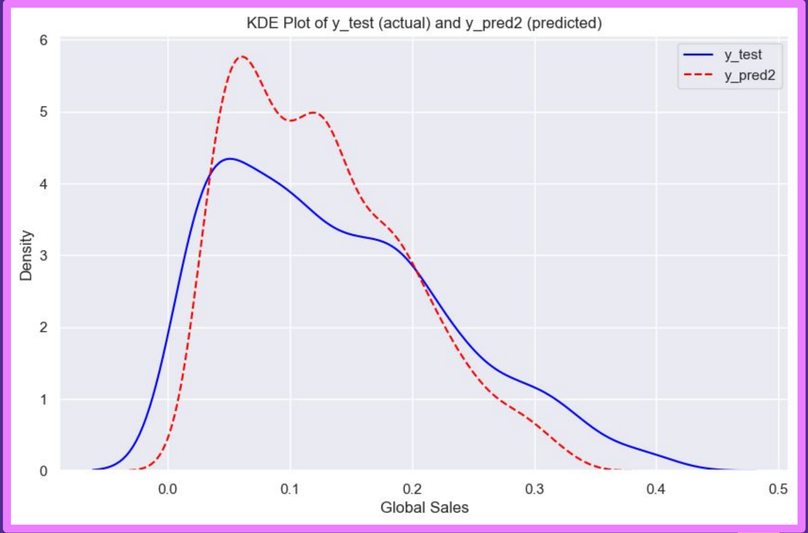
**Metrics for Lasso model with hyperparameter tuning:**

- MAE: 0.02777958365283314
- MSE: 0.0020869010634740002
- R^2 Score: 0.7564381553955781

**Intercept of Lasso model with best hyperparameters:**
0.1329005644811157

**Coefficients of Lasso model with best hyperparameters:**

0.04928594 0.0148108  0.01280174 0,  0,  0, 0, 0, 0, 0.00889946, 0



# Overall results of Lasso

- ★ Significant improvements in predictive performance following hyperparameter tuning
- ★ By adjusting the regularization parameter, the tuned model exhibited lower prediction errors and better captured the underlying patterns in the data
- ★ Improves predictive modeling accuracy and decision-making processes

# Radial Basis Function Network (RBF)

# What is Radial Basis Function Network?

1 **Definition**

Type of feed forward neural network composed of three layer,the input layer,the hidden layer and the output layer

2 **Function**

Increase the accuracy of prediction better the  other regression methods

3 **Equation**

$$y(\mathbf{x}) = \sum_{i=1}^{N} w_i \, \varphi(\|\mathbf{x} - \mathbf{x}_i\|),$$

# Process of coding RBF

Numerical Columns

Categorical Columns

All columns

# Code breakdown

1. Data preparation
2. Defining the functions for RBF
3. Defining the hyperparameters for tuning
4. Fitting the training data
5. Predicting in test data
6. Evaluating the model

```python
import numpy as np
from sklearn.base import BaseEstimator, RegressorMixin
from sklearn.cluster import KMeans
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler

class RBFNRegressor(BaseEstimator, RegressorMixin):
    def __init__(self, n_centers=10, gamma=1.0, alpha=1.0): #initializes the RBFNRegressor with default values
        self.n_centers = n_centers
        self.gamma = gamma
        self.alpha = alpha    #alpha gives regularization strength for Ridge Regression
        self.centers = None
        self.linear_model = None

    def fit(self, X, y):
        #Fits the RBFNRegressor to training data
        # Step 1: Initialize centers using K-Means clustering
        kmeans = KMeans(n_clusters=self.n_centers, random_state=42)  # Ensure reproducibility
        kmeans.fit(X)
        self.centers = kmeans.cluster_centers_

        # Step 2: Compute radial basis functions
        rbf = self._compute_rbf(X)

        # Step 3: Train linear model with RBF features
        self.linear_model = Ridge(alpha=self.alpha)
        self.linear_model.fit(rbf, y)

    def predict(self, X): #predicts target value for given input data using trained RBFNRegressor
        rbf = self._compute_rbf(X)
        return self.linear_model.predict(rbf)

    def _compute_rbf(self, X): #computes RBF features for input data based on centre centres
        rbf = np.array([np.exp(-self.gamma * np.linalg.norm(x - c) ** 2)
                        for x in X for c in self.centers])
        return rbf.reshape(len(X), self.n_centers)

    def get_params(self, deep=True):
        return {'n_centers': self.n_centers, 'gamma': self.gamma, 'alpha': self.alpha}

    def set_params(self, **parameters):
        for parameter, value in parameters.items():
            setattr(self, parameter, value)
        return self
```

# Results

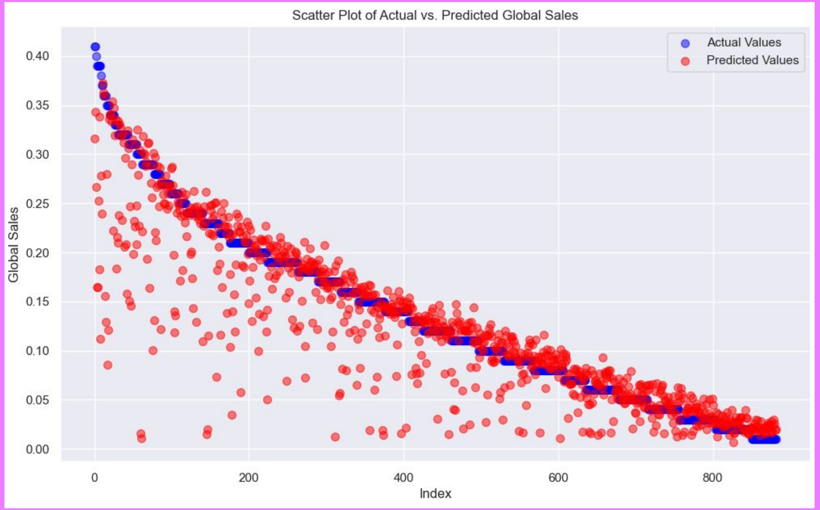
# Numerical Columns: Results

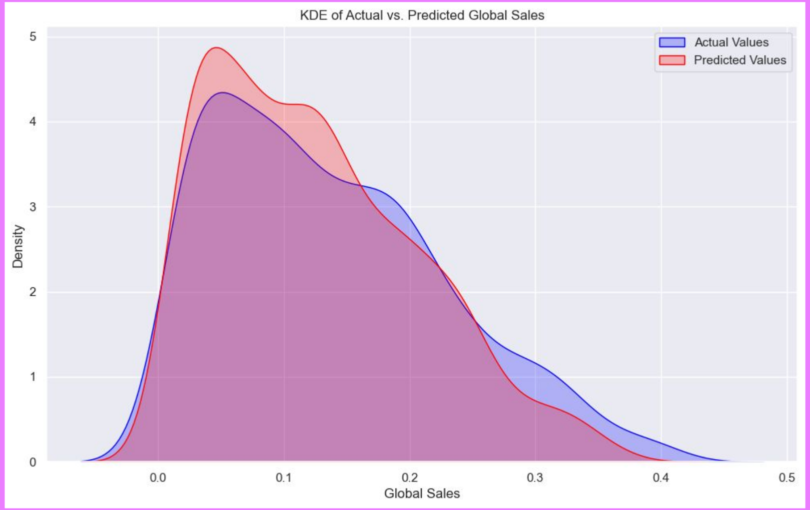
## Best Hyperparameters:

```
Best Hyperparameters: {'alpha': 0.0001, 'gamma': 0.001, 'n_centers': 40}
```

## Performance of Model:

**MSE:** 0.002203656669595725 **MAE:** 0.025192976277781668 **R^2 Score:** 0.742811629781769



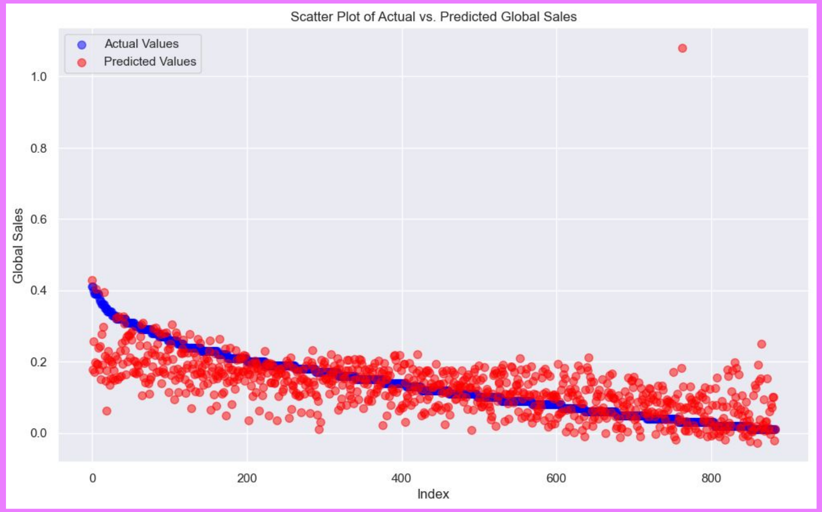
# Categorical Columns: Results

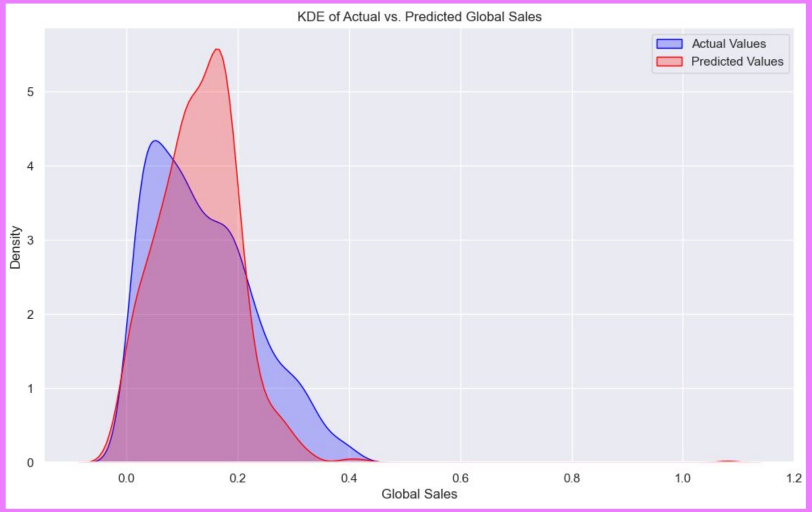
**Best Hyperparameters:**

```
Best Hyperparameters: {'alpha': 0.0001, 'gamma': 0.001, 'n_centers': 30}
```

**Performance of Model:**

**MSE:** 0.005907722580636701 **MAE:** 0.05300455005331339 **$R^2$ Score:** 0.3105107691325899



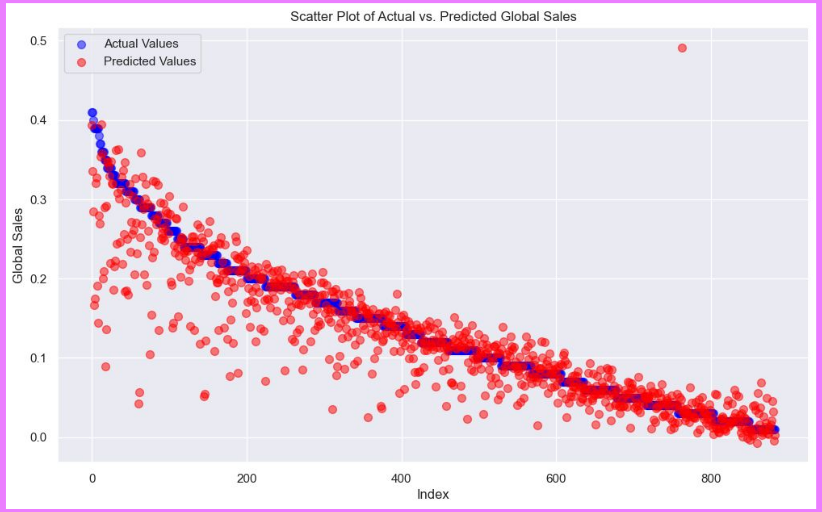
# Numerical and Categorical Columns: Results

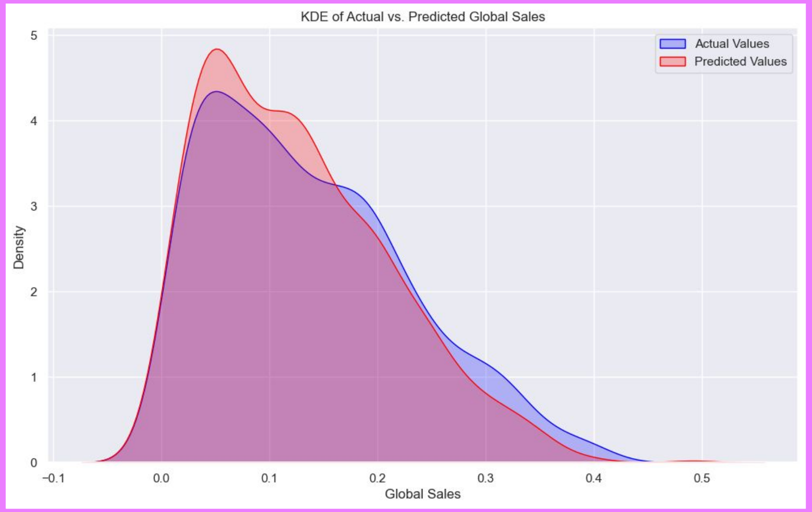
## Best Hyperparameters:

Best Hyperparameters: {'alpha': 0.0001, 'gamma': 0.001, 'n_centers': 60}

## Performance of Model:

**MSE:** 0.001976806785410951 **MAE:** 0.024543372597260001 **R^2 Score:** 0.7692872386198644



# Analysis from Models used

| | **Numeric** | Categorical | All |
|---|---|---|---|
| Best performance for Linear | **R^2 Score :0.54476**<br><br>**MSE: 0.00390** | | |
| Best performance for Lasso | **R^2 Score: 0.72475**<br><br>**MSE: 0.00236** | | |
| Best performance for RBF | **R^2 Score :0.72475**<br><br>**MSE:0.002203** | | |

R^2 for RBF & Lasso > Linear

MSE for RBF < Lasso & Linear

# Analysis from Models used

| | Numeric | **Categorical** | All |
|---|---|---|---|
| Best performance for Linear | R^2 Score :0.54476<br><br>MSE: 0.00390 | **R^2 Score :0.431708**<br><br>**MSE: 0.004869** | |
| Best performance for Lasso | R^2 Score: 0.72475<br><br>MSE: 0.00236 | **R^2 Score :**0.455976<br><br>**MSE:** 0.0046613 | |
| Best performance for RBF | R^2 Score :0.72475<br><br>MSE:0.002203 | **R^2 Score :**0.3105108<br><br>**MSE:** 0.0530046 | |

R^2: Lasso > RBF
MSE: Lasso < RBF

# Analysis from Models used

| | Numeric | Categorical | **All** |
|---|---|---|---|
| Best performance for Linear | R^2 Score :0.54476<br><br>MSE: 0.00390 | R^2 Score :0.431708<br><br>MSE: 0.004869 | - |
| Best performance for Lasso | R^2 Score: 0.72475<br><br>MSE: 0.00236 | R^2 Score :0.455976<br><br>MSE: 0.0046613 | **R^2 Score : 0.756438**<br><br>**MSE: 0.002087** |
| Best performance for RBF | R^2 Score :0.72475<br><br>MSE:0.002203 | R^2 Score :0.3105108<br><br>MSE: 0.0530046 | **R^2 Score :0.769287**<br><br>**MSE: 0.0019767** |

R^2 > 0.7 for Lasso & RBF

# OUTCOME

1. Best Regression Model: Lasso Regression
2. Best model for prediction (overall): RBF Networks
3. No significant improvement when predicted using RBF network
4. Best columns for numeric prediction: NA_Sales, EU_Sales, Other_Sales
5. Best columns for categorical prediction: mean_Developer (Developer), mean_Platform (Platform), mean_Genre (Genre)

# INSIGHTS

#Insight 1: Lasso > Linear

1. Performs **feature selection**, **reduces overfitting**, manages **multicollinearity**, **increases robustness** to outliers, enhances model **interpretability**

2. Lasso has L1 regularization

# INSIGHTS

#Insight 2: Lasso Linear ≅ RBF

1. RBF only shows slight improvement compared to Lasso

2. Due to hyperparameter tuning sensitivity, unnecessary complexity for linear r/s and data size issues

# INSIGHTS

#Insight 3: Predicted data and Real life example

1. USA and Europe are the biggest market for gaming

2. Other sales includes Asian countries

3. Genre: Game appeal to user

4. Platform & Developer: Compatibility & Accessibility

| | | | | |
|---|---|---|---|---|
| 1. | | United States | $46.4B | 209.8M |
| 2. | | China | $44.0B | 696.5M |
| 3. | | Japan | $19.1B | 73.4M |
| 4. | | South Korea | $7.4B | 33.3M |
| 5. | | Germany | $6.5B | 49.5M |
| 6. | | United Kingdom | $5.5B | 38.5M |

# THANKS!