Faculty Of Engineering
Ain Shams University

CSE472-Artificial Intelligence

Artificial Intelligence Project

Team Members: -

- Abdelrahman Mostafa Ali El Deen 22P0150
- Mohamed Ashraf Mohamed 22P0210
- Mohamed Basher Mohamed 22P0223
- Omar Ahmed Abdelmohsen 22P0218
- Roger Sherif Selim Salama 22P0112
- Seif Aly Othman Fahmy 22P0182

Presented To: -

- Dr. Mariam Nabil El Berri
- Eng. Mohamed Essam

Acknowledgement: -

We would like to express Our sincere gratitude to **Dr. Mariam** for her exceptional guidance, dedication, and support throughout our Artificial Intelligence course. Her deep knowledge and passion for the subject have greatly enriched our learning experience and inspired us to explore the field more deeply.

We would also like to thank our **Teaching Assistant, Mohamed**, for his continuous assistance, clear explanations, and patience. His efforts in helping us understand complex topics and ensuring our progress did not go unnoticed.

Together, Dr. Mariam and TA Mohamed have made this journey into Artificial Intelligence, both insightful and rewarding, and we are truly grateful for the opportunity to learn from them.

Contents

1.0 Introduction: -

In this project, we aim to build a system that predicts the rating of applications using machine learning techniques. We are provided with a dataset containing various attributes related to mobile applications. The project involves training four different models to predict app ratings based on this data.

Our process begins with cleaning the dataset to handle missing values, outliers, and inconsistencies. We then perform feature selection to identify the most relevant variables for training. Additionally, we calculate some descriptive statistics to better understand the dataset and its structure.

After preprocessing, we train the four models and evaluate their performance using a separate test dataset. Once the models make their predictions, we measure their accuracy and compare their results to determine which model performs best in predicting application ratings.

2.0 Data Cleaning: -

The first thing we notice is that there are 11 features, all using the object datatype, and the feature we are trying to predict has many nulls. We will either drop the rows with null values in Y or replace with the median Y.

```
df = pd.read_csv("/kaggle/input/app-rating-competition/train.csv")
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8968 entries, 0 to 8967
Data columns (total 13 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   X0      8968 non-null   object
 1   X1      8968 non-null   object
 2   X2      8968 non-null   object
 3   X3      8968 non-null   object
 4   X4      8968 non-null   object
 5   X5      8967 non-null   object
 6   X6      8968 non-null   object
 7   X7      8967 non-null   object
 8   X8      8968 non-null   object
 9   X9      8968 non-null   object
 10  X10     8961 non-null   object
 11  X11     8965 non-null   object
 12  Y       7494 non-null   float64
dtypes: float64(1), object(12)
memory usage: 910.9+ KB
```

+ Code + Markdown

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	Y
2	Command	FAMILY	0	Varies with dev	0		0	Everyone	Strategy	June 28, 2(	Varies with	Varies with device	
3	Popsicle La	PERSONAL	0	5.5M	0+	Paid	$1.49	Everyone	Personaliz	July 11, 20	1.1	4.2 and up	
4	Ak Parti Ya	SOCIAL	0	8.7M	0+	Paid	$13.99	Teen	Social	July 28, 20	3.4.4.3.3	4.1 and up	
5	AP Series S	FAMILY	0	7.4M	0+	Paid	$1.99	Everyone	Education	July 30, 20	1.3	4.0 and up	
6	Ain Arabic	FAMILY	0	33M	0+	Paid	$2.99	Everyone	Education	April 15, 2(	1	3.0 and up	
7	cronometr	PRODUCTI	0	5.4M	0+	Paid	$154.99	Everyone	Productivit	November	1.0.0	4.1 and up	
8	Pekalonga	SOCIAL	0	5.9M	0+	Free		0 Teen	Social	July 21, 20	0.0.1	4.4 and up	
9	CX Networ	BUSINESS	0	10M	0+	Free		0 Everyone	Business	August 6, 2	1.3.1	4.1 and up	
10	Sweden Ne	NEWS_AN	0	2.1M	0+	Free		0 Everyone	News & M	July 7, 201	1.1	4.4 and up	
11	Test Applic	ART_AND_	0	1.2M	0+	Free		0 Everyone	Art & Desi	March 14,	4	4.2 and up	
12	EG | Explo	TRAVEL_A	0	56M	0+	Paid	$3.99	Everyone	Travel & L	January 22	1.1.1	4.1 and up	
13	EP Cook B	MEDICAL	0	3.2M	0+	Paid	$200.00	Everyone	Medical	July 26, 20	1	3.0 and up	
14	Eu sou Ric	FINANCE	0	2.6M	0+	Paid	$30.99	Everyone	Finance	January 9,	1	4.0 and up	
15	Eu Sou Ric	FINANCE	0	1.4M	0+	Paid	$394.99	Everyone	Finance	July 11, 20	1	4.0.3 and up	
16	I'm Rich/E	LIFESTYLE	0	40M	0+	Paid	$399.99	Everyone	Lifestyle	December	MONEY	4.1 and up	
17	Subway Su	GAME	27722264	76M	1,000,000,	Free		0 Everyone	Arcade	July 12, 20	1.90.0	4.1 and up	4.5
18	Subway Su	GAME	27724094	76M	1,000,000,	Free		0 Everyone	Arcade	July 12, 20	1.90.0	4.1 and up	4.5
19	Instagram	SOCIAL	66577313	Varies with dev	1,000,000,	Free		0 Teen	Social	July 31, 20	Varies with	Varies with device	4.5
20	Instagram	SOCIAL	66577313	Varies with dev	1,000,000,	Free		0 Teen	Social	July 31, 20	Varies with	Varies with device	4.5
21	Google Ph	PHOTOGR	10858556	Varies with dev	1,000,000,	Free		0 Everyone	Photograp	August 6, 2	Varies with	Varies with device	4.5
22	Google Ph	PHOTOGR	10858538	Varies with dev	1,000,000,	Free		0 Everyone	Photograp	August 6, 2	Varies with	Varies with device	4.5
23	Google Ph	PHOTOGR	10859051	Varies with dev	1,000,000,	Free		0 Everyone	Photograp	August 6, 2	Varies with	Varies with device	4.5
24	Subway Su	GAME	27711703	76M	1,000,000,	Free		0 Everyone	Arcade	July 12, 20	1.90.0	4.1 and up	4.5
25	Instagram	SOCIAL	66509917	Varies with dev	1,000,000,	Free		0 Teen	Social	July 31, 20	Varies with	Varies with device	4.5
26	Google Ph	PHOTOGR	10847682	Varies with dev	1,000,000,	Free		0 Everyone	Photograp	August 1, 2	Varies with	Varies with device	4.5

All features require cleaning, and some require replacing "Varies with device" values to actual values with either most frequent imputation or median. We can notice by simply observing the data set that not all columns will be used in our model training, for example columns X0 logically plays no role in the rating that an application would get. Columns X5 and X6 are related and dependant on each other so one column will be dropped as to avoid higher accuracy error. Furthermore, we drop column X10 as it is very ambiguous and it is very hard to determine what it is used for.

```
df = df.drop(columns=['X8', 'X0','X10'])
```

```
df_clean = df_clean.dropna(subset=['Y'])
df_clean = df_clean[(df_clean['Y'] >= 1) & (df_clean['Y'] <= 5)]
#df_clean['Y'] = df_clean['Y'].fillna(df_clean['Y'].median())
```

We start off with removing all rows where Y is below 1 or above 5, then we either remove null values or replace them with median.

We have the two options and switched between them during validation.

2.1 Handling Features:

2.1.1 Column X2:

```
# Preprocess X2 (Reviews)
df_clean['X2'] = df_clean['X2'].astype(str).str.replace(',', '', regex=False)
df_clean['X2'] = pd.to_numeric(df_clean['X2'], errors='coerce')
```

The column X2, which likely contains review counts, is first converted to strings and stripped of commas (e.g., "1,000" becomes "1000") using str.replace(). After that, it's converted to a numeric type with pd.to_numeric(), using the 'coerce' option to handle any values that can't be converted by replacing them with NaN.

2.1.2 column X3:

```
# Convert size strings like '10M', '500K' to float in MB
def convert_size(value):
    if pd.isna(value):
        return pd.NA
    value = str(value)
    if 'M' in value:
        return float(value.replace('M', ''))
    elif 'K' in value or 'k' in value:
        return float(value.replace('K', '').replace('k', '')) / 1024
    return pd.NA
```

This function convert size, is used to convert size strings such as "10M" or "500K" into floating-point numbers representing the size in

megabytes (MB). It checks if the input is null using pd.isna() and returns a missing value (pd.NA) if so. Otherwise, it converts the input to a string and then determines if the string includes 'M' (indicating megabytes) or 'K'/'k' (indicating kilobytes). For 'M', it strips the letter and converts the remaining number directly to a float. For 'K', it strips the letter(s) and divides the resulting number by 1024 to convert kilobytes to megabytes.

```
# Process X3 (Size)
df_clean['X3'] = df_clean['X3'].replace('Varies with device', pd.NA)
df_clean['X3'] = df_clean['X3'].apply(convert_size)
df_clean['X3'] = pd.to_numeric(df_clean['X3'], errors='coerce')
```

In this function any instance of the string "Varies with device" is replaced with a missing value (pd.NA). The custom function convert_size is then applied to convert strings like "10M" or "500K" into floating-point numbers representing size in megabytes. Afterward, the column is converted to numeric format for further analysis.

2.1.3 Column X4:

```
# Process X4 (Downloads)
df_clean['X4'] = df_clean['X4'].apply(lambda x: x.strip('+').replace(',', ''))
df_clean['X4'] = pd.to_numeric(df_clean['X4'], errors='coerce')
```

In column X4, assumed to be the number of downloads, the apply () function uses a lambda to remove plus signs (+) and commas from the values. For instance, "1,000+" becomes "1000". This cleaned data is then coerced into numeric values.

2.1.4 Column X5:

```
# Process X5 (Type: Free=0, Paid=1)
df_clean['X5'] = df_clean['X5'].map({'Free': 0, 'Paid': 1})
```

Column X5 appears to represent the app type — either "Free" or "Paid". These string values are mapped directly to integers using a dictionary, where "Free" becomes 0 and "Paid" becomes 1.

2.1.5 Column X6:

```
 # Process X6 (Type: Free=0, Paid=1)
df_clean['X6'] = pd.to_numeric(df_clean['X6'].str.replace('$', '', regex=False), errors='coerce')
```

Column X6, representing the price of the app, has dollar signs ($) stripped from the values. The result is then converted to numeric format. If any non-numeric entries remain, they are coerced into missing values.

2.1.6 Column X9:

```
def parse_date(value):
    try:
        # Convert to datetime, handling common date formats
        return pd.to_datetime(value, errors='coerce', format='%B %d, %Y')
    except Exception as e:
        return pd.NaT  # Return Not a Time for invalid dates
```

The second function, parse_date, attempts to convert a string value into a datetime object using pandas.to_datetime(). It specifies a common date format ('%B %d, %Y') and uses errors='coerce' to ensure that any invalid dates result in a missing timestamp (pd.NaT) instead of an exception. If the conversion fails for any reason, the function catches the exception and returns pd.NaT to gracefully handle invalid or malformed dates.

2.1.7 Column X11:

```
# Extract float version from a string like '4.1 and up'
def preprocess_X11(value):
    if pd.isna(value) or 'Varies' in str(value):
        return pd.NA
    try:
        return float(str(value).split()[0])
    except ValueError:
        return pd.NA
```

This function is designed to extract the numeric part of a string like "4.1 and up", which might represent version numbers or similar values. It first checks if the value is null or contains the word "Varies" and returns pd.NA in such cases. Otherwise, it tries to convert the first part of the string (before any spaces) into a float, effectively extracting the numeric version. If this conversion fails, a ValueError is caught, and a missing value is returned.

```
# Process X11
df_clean['X11'] = df_clean['X11'].apply(preprocess_X11)
df_clean['X11'] = pd.to_numeric(df_clean['X11'], errors='coerce')
```

Finally, column X11, which may represent Android version requirements or similar versioning information, is processed with the previously defined preprocess_X11 function. This function extracts the numeric portion of strings like "4.1 and up". The result is then converted into numeric format, enabling use in models or statistical analysis.

```
import pandas as pd

# Load the CSV file
df = pd.read_csv('train.csv')

# Remove rows with empty values in the 'Y' column
df = df[df['Y'].notna()]

# Convert 'Y' column to numeric (in case it's stored as strings)
df['Y'] = pd.to_numeric(df['Y'])

# Define the ranges for grouping the values
bins = [0, 1, 2, 3, 3.5, 4, 4.1, 4.3, 4.5, 4.7, 4.9, 5]
labels = ['0-1', '1-2', '2-3', '3-3.5', '3.5-4', '4-4.1',
          '4.1-4.3', '4.3-4.5', '4.5-4.7', '4.7-4.9', '4.9-5']

# Categorize the values into the defined ranges
df['Range'] = pd.cut(df['Y'], bins=bins, labels=labels, include_lowest=True)

# Calculate the percentage distribution
percentage_distribution = df['Range'].value_counts(normalize=True) * 100

# Sort the results by the ranges for better presentation
percentage_distribution = percentage_distribution.reindex(labels)

# Print the results
print("Percentage distribution of values in the 'Y' column:")
print(percentage_distribution)
```

This code takes the Y column and drops the null values from it , divides the data into ranges and returns the percentages of these ranges.

2.2 Feature Engineering: -

Two new engineered features were added:

1. Reviews per downloads
2. Days since date

```
df_clean['X12'] = np.where(
    (df_clean['X4'] == 0) | (df_clean['X4'].isna()),
    0,
    df_clean['X2'] / df_clean['X4']
)
# processing date X9
df_clean['X9'] = df_clean['X9'].apply(parse_date)
today_date = datetime( year: 2025, month: 5, day: 8)
# Calculate the number of days since today (08/05/2025)
df_clean['X13'] = (today_date - df_clean['X9']).dt.days
```

First, we create a new column X12 by dividing values from column X2 by those in X4, but only when X4 contains valid, non-zero numbers. If X4 is zero or missing (NaN), the result defaults to zero to avoid division errors. After this calculation, a logarithmic transformation (log1p) is applied to X12 to normalize its distribution, which is useful for handling skewed data.

Next, the code processes dates stored in column X9 by converting them into datetime objects using a function called parse_date. It then calculates the time elapsed between these dates and a fixed reference date (May 8, 2025), storing the difference in days as a new column X13. This feature could represent the age of an item, the time since an event, or another time-based metric.

Later, these engineered features will be correlated with the rating and checked if it could improve predictions.

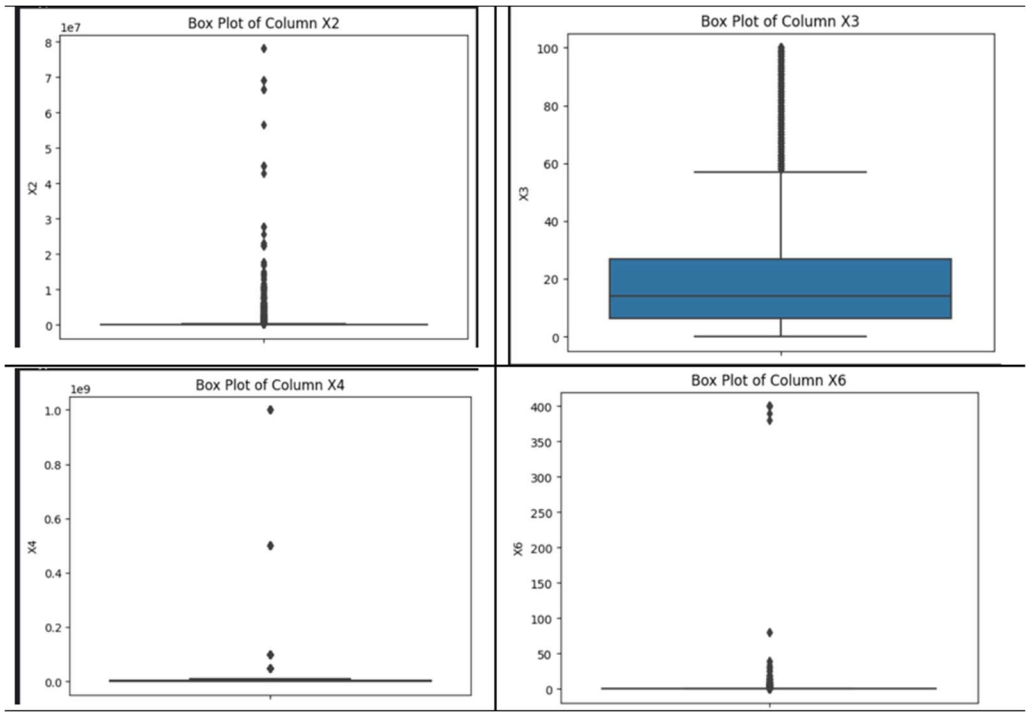
Before continuing, we noticed there were some null values in X3 and X11, so we used the imputer to replace them with the most frequent values.

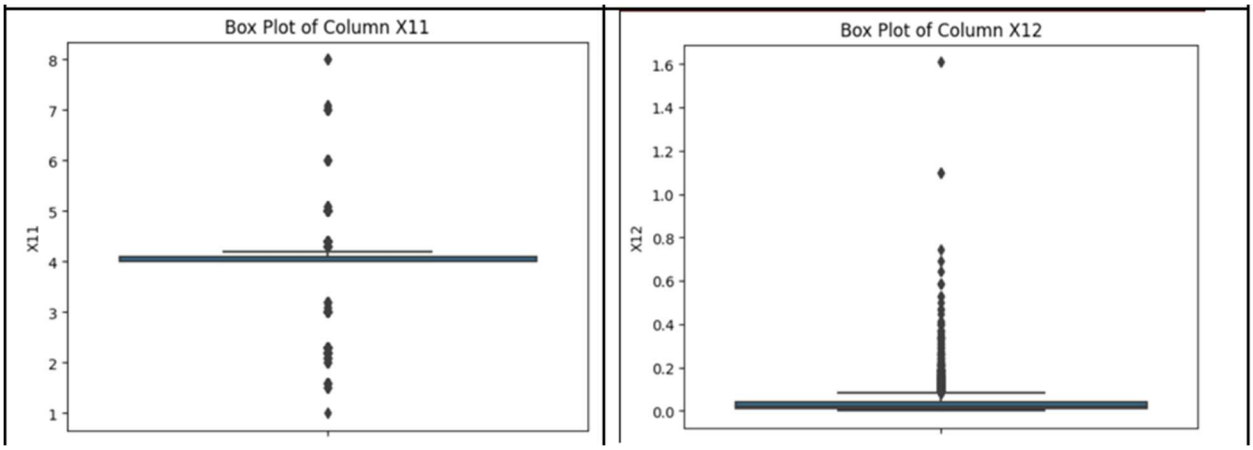
```
X1        0
X2        0
X3     1301
X4        0
X5        0
X6        0
X7        0
X9        0
X11    2221
Y         0
X13       0
dtype: int64
```

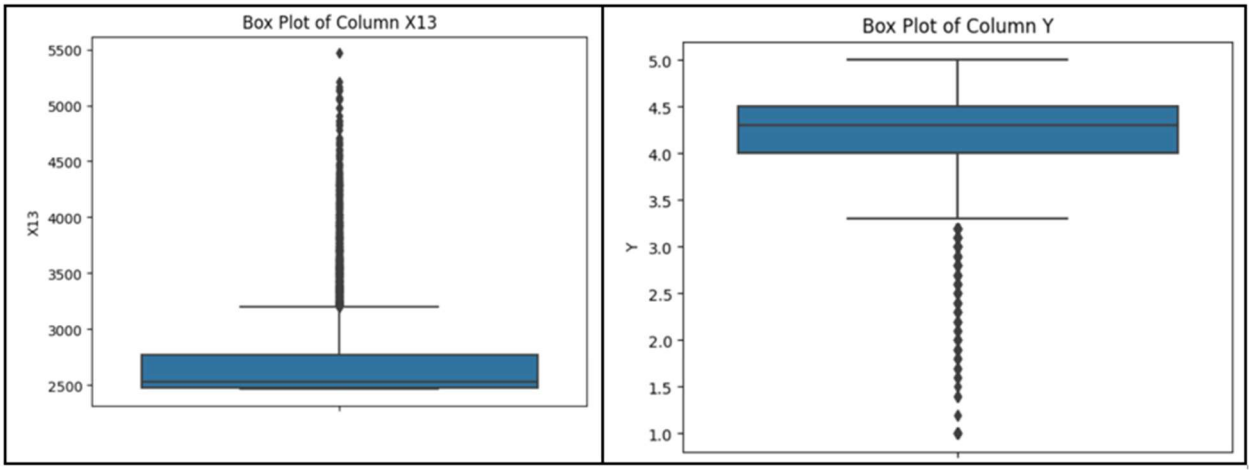
```python
# Impute missing values
imputer = SimpleImputer(strategy='most_frequent')
df_clean[['X3','X11']] = imputer.fit_transform(df_clean[['X3','X11']])
```

We may need to change the strategy of the imputer later to get better results, we kept that in mind.

We plotted Box plots of most features to take note of the outliers, as they should be dropped when using models that are sensitive to outliers.







The outliers are considerably large, so we tried removing them with the following function.

```python
def remove_outliers_iqr(df, columns):
    for col in columns:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        df = df[(df[col] >= lower_bound) & (df[col] <= upper_bound)]
    return df

# Apply to X2, X4, and X13
df_clean = remove_outliers_iqr(df_clean, ['X2', 'X4', 'X13'])
```

This function was tested on many models and features and always gave worse results, so we concluded that the outliers were informative, not noise.

2.3 Data Validation (k-folds):

K-Fold Cross-Validation (CV) is a fundamental technique used in machine learning to evaluate model performance more reliably than a simple train-test split. Our code implements this method by first dividing the dataset into multiple subsets called folds - by default using 5 folds, as specified in the function parameters. Each fold takes turns serving as the validation set while the remaining folds are used for training, ensuring that every data point contributes to both training and validation exactly once. This rotation process provides a comprehensive assessment of the model's predictive capabilities across different subsets of data.

The implementation begins by initializing the KFold object with three key parameters: the number of splits (n_splits), a shuffle option set to True to randomize the data before splitting, and a fixed random state (80) for reproducibility. During each iteration of the cross-validation loop, the data is partitioned into training and validation sets using indices generated by the KFold splitter. The model is then trained on the scaled training data and evaluated on the validation set, with performance metrics recorded for each fold.

This approach offers several advantages for machine learning projects. First, it provides a more robust estimate of model performance by averaging results across multiple data partitions, reducing the variance that might occur from a single random train-test split. Second, it makes efficient use of limited data by ensuring all observations contribute to both training and evaluation. Third, the shuffling of data before splitting helps prevent any inherent ordering in the dataset from influencing the results. The final output includes average scores for R-squared, MAE, and RMSE across all folds, giving a comprehensive view of the model's predictive accuracy and error magnitude.

3.Used Models: -

3.1 Linear regression:

Linear Regression is a **supervised machine learning algorithm** used for predicting a continuous numerical output (like your application ratings) based on one or more input features. It assumes a **linear relationship** between the input variables (features) and the output (target).

- **Model Representation**:
  1- The algorithm tries to fit a straight line (in simple regression) or a hyperplane (in multiple regression) that best predicts the target variable.
  2- The equation for a simple linear regression is:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \epsilon$$

  - y = predicted rating (dependent variable)
  - $x1$, $x2$, …, xn = input features (independent variables)
  - $\beta 0$ = y-intercept (bias term)
  - $\beta 1$, $\beta 2$, …, βn = coefficients (weights)
  - ϵ = error term (difference between predicted and actual value)

- **Training Process**:
  1- The model **learns** the best coefficients (β) by minimizing the **Mean Squared Error (MSE)** between predicted and actual values.
  2- Optimization techniques like **Ordinary Least Squares (OLS)** or **Gradient Descent** adjust the coefficients to find the best-fit line.

Unlike complex models (e.g., neural networks), linear regression provides clear coefficients, showing how each feature affects the predicted rating. Also, it serves as a **benchmark** to compare more advanced models (e.g., Random Forest, Gradient Boosting). Finally, it trains quickly, even on moderately large datasets, making it suitable for initial testing.

Features Used: -

1) X3
2) X5
3) X11
4) X7

However, X7 was different as it was a string that states which age group the app was suitable for. In the case of the linear regression model **One Hot Encoding** was performed where 5 columns where each column is a different age group specified by X7 e.g. (a column for adults, a column for everyone, etc.) in these columns there were only ones and zeros as to turn the strings of X7 to Boolean where a column will contain 1 in the rows where the application is suitable for this category e.g.

3.2 KNN Model:

The **K-Nearest Neighbours Regressor (KNN Regressor)** is a **non-parametric, instance-based** machine learning algorithm used for **predicting continuous values**. Unlike models that learn a set of weights or coefficients, KNN doesn't "learn" in the traditional sense — it simply **stores the training data** and makes predictions based on proximity.

How it works:

1. **Distance Calculation**:
    - When predicting the target value (Y) for a new data point, KNN computes the **distance** (Euclidean) between this point and all the data points in the training set.
    - In this project, the distances are computed based on two numerical features:
        - **X12**: Ratio of reviews to downloads.
        - **X13**: Days since app release.
2. **Find Neighbours**:
    - The algorithm identifies the **k closest data points** (in our case, k=88) from the training set — these are the "neighbours."
3. **Prediction**:
    - The prediction is the **average of the Y values** of the k nearest neighbours.
    - Since this is regression (not classification), it returns a continuous number instead of a category.

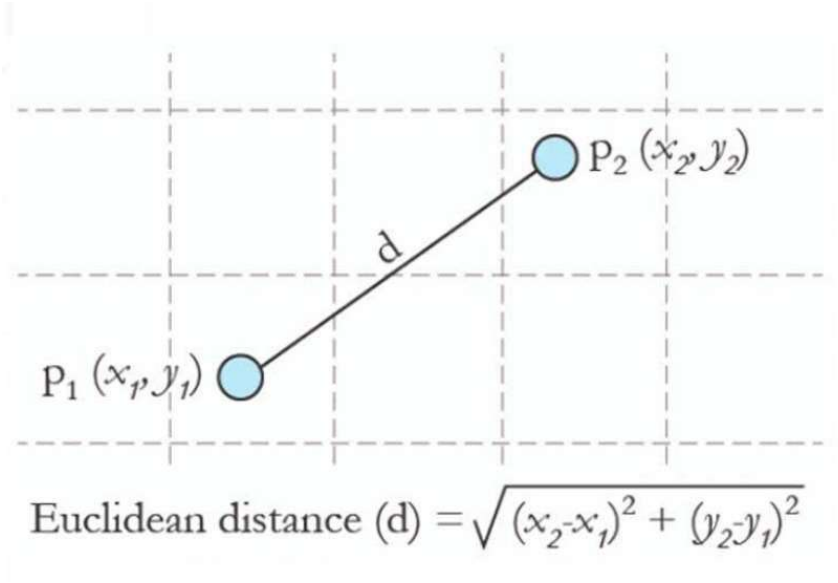
I used **k = 88** in the KNN Regressor because it is approximately the **square root of the number of data points** used for training, which was **7,900 rows** after cleaning the dataset. This is a common heuristic in KNN to balance bias and variance — using too small a k can lead to noisy predictions, while too large a k can overly smooth the output. Choosing $\sqrt{n}$ helps provide a reasonable trade-off for many datasets.

Why these features were used:

- **X12 (reviews/downloads)**: This feature represents a normalized engagement level. A high number of reviews per download can suggest high user involvement or satisfaction.
- **X13 (days since release)**: This reflects the app's maturity. Older apps may have more stable ratings due to long-term user feedback, while new apps may have fluctuating or fewer ratings.

These features were selected because:

- Both are **numerical and continuous**, which fits well with KNN's reliance on distance computations.
- They are **strongly correlated** with the target variable Y (as shown by the Pearson correlation), suggesting they are informative.
- They are **independent of categorical variables**, which could introduce complexity in KNN without proper encoding.



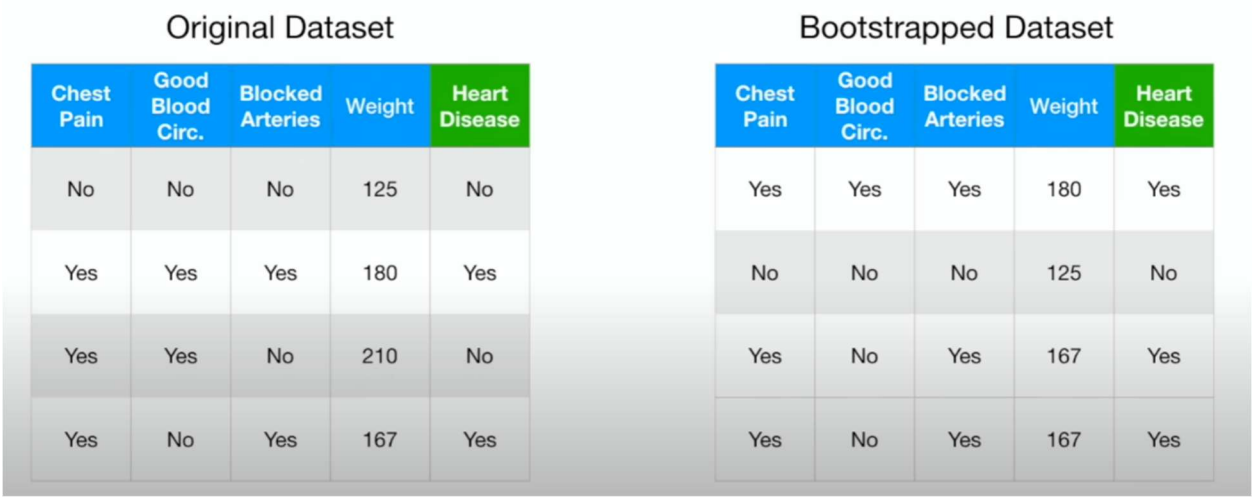
3.3 Random Forest Model:

The machine learning model employed in this project is a **Random Forest Regressor**, a robust ensemble learning method designed for regression tasks. Random Forest operates by constructing multiple decision trees during training and outputs the mean prediction of the individual trees. This method effectively mitigates overfitting commonly associated with individual decision trees by leveraging ensemble learning to enhance generalization, but how does it work?

Steps for Random Forest:

1) **Bootstrap Sampling:**

During training, Random Forest uses a technique called bootstrap sampling to create multiple subsets of the training



Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
No	No	No	125	No
Yes	Yes	Yes	180	Yes
Yes	Yes	No	210	No
Yes	No	Yes	167	Yes

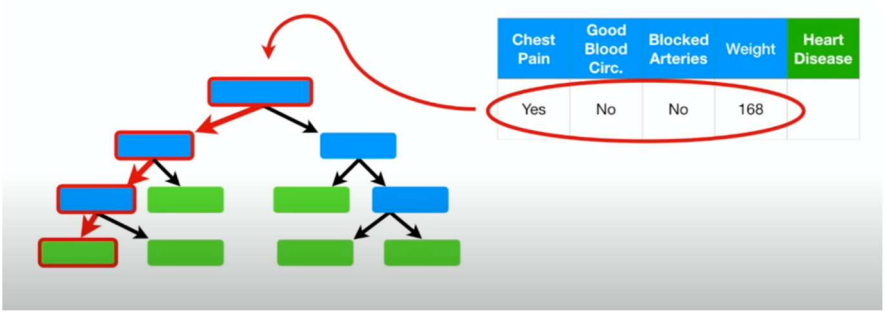
Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
Yes	Yes	Yes	180	Yes
No	No	No	125	No
Yes	No	Yes	167	Yes
Yes	No	Yes	167	Yes

data. Each subset is generated by randomly sampling the original data with replacement. This means some data points may appear multiple times in a subset, while others may be excluded. Each subset is used to train an individual decision tree.

2) **Decision Tree Training:**

Each decision tree in the forest is trained independently on its corresponding bootstrap sample. Unlike traditional decision

trees, Random Forest introduces randomness in the splitting process by selecting a random subset of features (specified by the max_features parameter) to consider at each node. This randomness ensures that the trees are diverse, reducing the risk of overfitting to the training data.

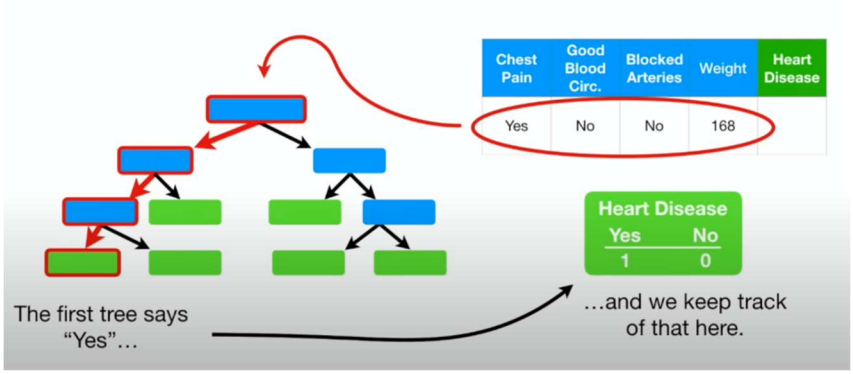


Node Splitting:

At each node of a decision tree, the algorithm evaluates potential splits using the randomly selected subset of features. It chooses the split that minimizes the error (for regression tasks, this is often measured using metrics like mean squared error). This process continues recursively until one of the stopping criteria is met, such as reaching the max_depth or having fewer than min_samples_leaf samples in a node.

3) Tree Prediction:

Once trained, each decision tree can independently predict an output for a given input. In the case of regression, the prediction is a numerical value based on the



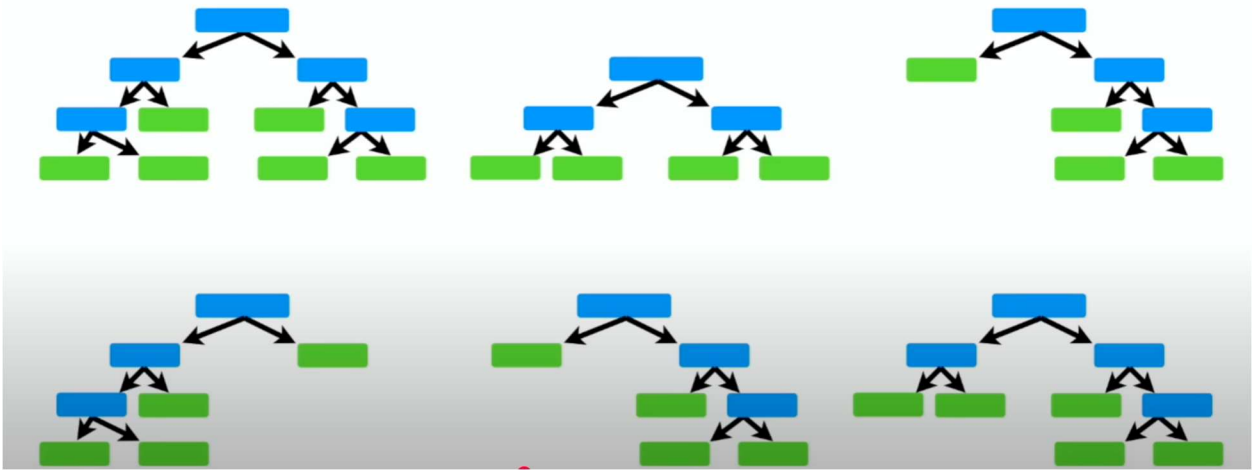
mean outcome of the training data that reaches the same leaf node as the input.

4) Aggregation of Predictions:

After all the trees have made their predictions, the Random Forest aggregates these results to produce the final output. For regression tasks, the ensemble prediction is the average of all the individual tree predictions. This averaging reduces the variance and increases the stability of the model, leading to better generalization on unseen data.

Hyperparameter Tuning:

The Random Forest model in this implementation is configured with specific hyperparameters tailored to optimize its predictive accuracy. The number of trees (**n_estimators**) is set to 200, meaning that the ensemble comprises 200 decision trees. This ensures sufficient variability in the individual models, improving the robustness and reducing the variance of the predictions.



The **max_depth** parameter is fixed at 6, limiting the maximum depth of each decision tree. This constraint helps prevent overfitting by

ensuring that the trees do not become overly complex and overly fitted to the training data.

Additionally, the **min_samples_leaf** parameter is set to 10, ensuring that a minimum of ten samples is present in the leaf nodes of the trees. This further enhances the generalization capability by avoiding splits that result in highly specific patterns.

```
# Train the model
model = RandomForestRegressor(n_estimators=200, random_state=42, min_samples_leaf=10,max_features=3,max_depth=6)
model.fit(X_train_scaled, y_train)
predictions = model.predict(X_test_scaled)
```

The **max_features** parameter is set to 3, which restricts the number of features considered for splitting at each node. This hyperparameter selection introduces randomness into the model-building process and enhances the model's ability to generalize by reducing correlation among the trees. Furthermore, a fixed random seed (**random_state**=42) is used to ensure reproducibility of results, maintaining consistency across different runs of the model.

5) Prediction on New Data:

The dataset contained missing values in features X3 and X11. Rather than discarding these features or imputing them with simple statistical values, a more informed approach was used. A separate Linear Regression model was trained to predict the missing values based on other available features. Once the missing values were filled using this model, the updated dataset was used to train the Random Forest Regressor. This step ensured that potentially informative features like X3 and X11 could be retained and leveraged by the model to improve its predictions.

```
 # === Impute X3 using Linear Regression on ['X4','X5','X9'] ===
mask_x3 = df_clean['X3'].isna()
if mask_x3.any():
    lr_x3 = LinearRegression()
    known = ~mask_x3
    lr_x3.fit(df_clean.loc[known, ['X4','X5','X9']],
              df_clean.loc[known, 'X3'])
    df_clean.loc[mask_x3, 'X3'] = lr_x3.predict(df_clean.loc[mask_x3, ['X4','X5','X9']])


# === Impute X11 using Linear Regression on ['X3','X4','X5','X9'] ===
mask_x11 = df_clean['X11'].isna()
if mask_x11.any():
    lr_x11 = LinearRegression()
    known11 = ~mask_x11
    lr_x11.fit(df_clean.loc[known11, ['X4','X3','X5','X9']],
               df_clean.loc[known11, 'X11'])
    df_clean.loc[mask_x11, 'X11'] = lr_x11.predict(df_clean.loc[mask_x11, ['X4','X3','X5','X9']])
```

Additionally, to ensure that the features are on a comparable scale before feeding them into the model, standardization is applied using a StandardScaler. This preprocessing step transforms the features to have a mean of 0 and a standard deviation of 1, which is particularly beneficial for algorithms like Random Forest that are sensitive to the relative scales of the input variables.

```
# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

When predicting outcomes for new data, the input features are passed through each decision tree in the forest. Each tree independently computes its prediction, and these results are aggregated to produce the final prediction. The ensemble's strength lies in combining the outputs of weak, diverse learners to create a more robust and accurate prediction.

6) Random Forest Modeling Approach and Optimization

Before settling on the final configuration, multiple hyperparameter combinations were tested to improve the model's predictive performance. Parameters such as the number of trees (n_estimators), maximum depth of each tree (max_depth), minimum number of samples per leaf (min_samples_leaf), and number of features considered at each split (max_features) were systematically varied. For each configuration, standard regression metrics including R-squared (R^2), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE) were computed. The version of the model that achieved the best performance on the leaderboard—indicating the best generalization to the test set—was selected as the final model.

```
    # Evaluate the model
    r2 = r2_score(y_test, predictions)
    mae = mean_absolute_error(y_test, predictions)
    rmse = mean_squared_error(y_test, predictions, squared=False)

    print(f"R²: {r2:.4f}, MAE: {mae:.4f}, RMSE: {rmse:.4f}")

    # Store the scores
    r2_scores.append(r2)
    mae_scores.append(mae)
    rmse_scores.append(rmse)

# Print the average scores
print("\nOverall Performance Across Folds:")
print(f"Mean R²: {np.mean(r2_scores):.4f}")
print(f"Mean MAE: {np.mean(mae_scores):.4f}")
print(f"Mean RMSE: {np.mean(rmse_scores):.4f}")
```

3.4 CatBoost Regression:

The CatBoost Regressor is a gradient boosting algorithm specifically designed to handle heterogeneous data, including both numerical and categorical features, with minimal preprocessing. It builds an ensemble of decision trees in a stage-wise fashion, optimizing a loss function (mean squared error for regression) at each step. In the following section we will talk about how the model works.

3.4.1 Ordered Boosting & Categorical Handling:

CatBoost employs a special scheme called *ordered boosting* which reduces target leakage when computing leaf statistics.

Since one-hot encoding blows up dimensionality for high-cardinality categories, catboost applies efficient permutation-driven encoding for categorial features that replaces each category with a statistic (e.g. mean target) computed on prior data only, eliminating the need for one-hot encoding.

3.4.2 Tree construction:

- **Iterations**: The model is trained for a fixed number of boosting rounds (iterations=500).
- **Depth**: Each tree has a maximum depth (depth=6), controlling its complexity.
- **Learning Rate**: A shrinkage factor (learning_rate= 0.000455) scales each tree's contribution, trading off speed of learning vs. risk of overfitting.

3.4.3 Ensembling:

At each iteration, a new decision tree is fit to the gradient of the loss function with respect to the current ensemble's predictions. The new tree's predictions are added (scaled by the learning rate) to the ensemble, gradually improving fit on hard-to-predict observations.

Unlike other traditional tree based models, which compute each tree separately and computes the average for them, catboost works sequentially where each tree corrects the previous errors.

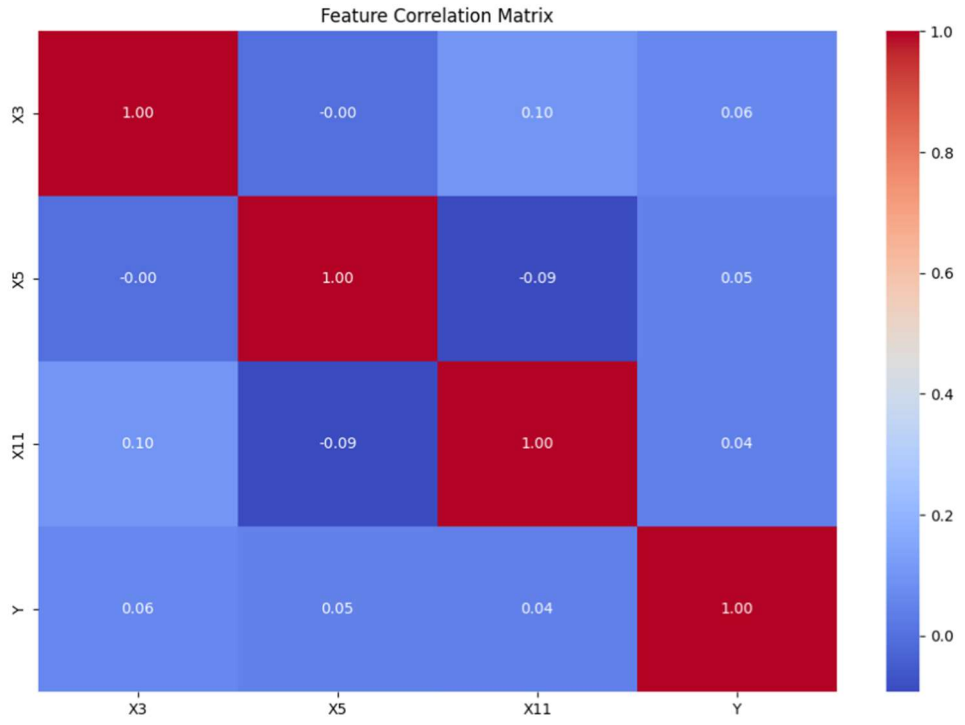
3.4.4 Prediction

For a new sample, each tree outputs a real-valued prediction; these are summed (with learning-rate scaling) to produce the final continuous target estimate.

3.4.5 Chosen Features

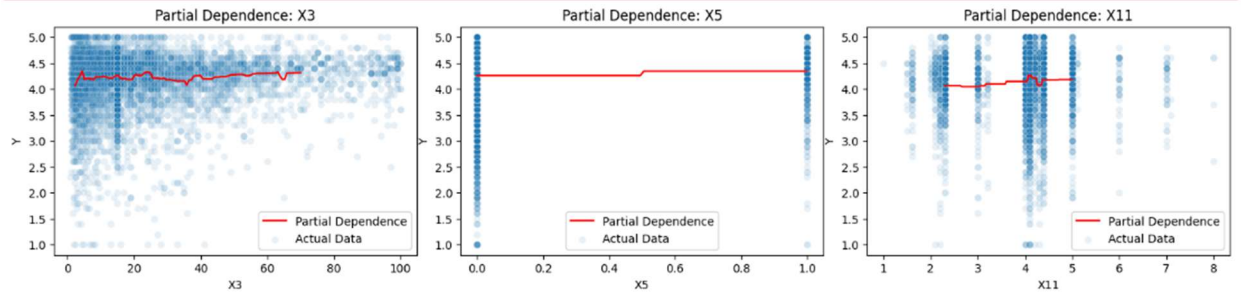
The final features used by our model were:

X[3] App size X[5] Free/Paid X[11] Device Version



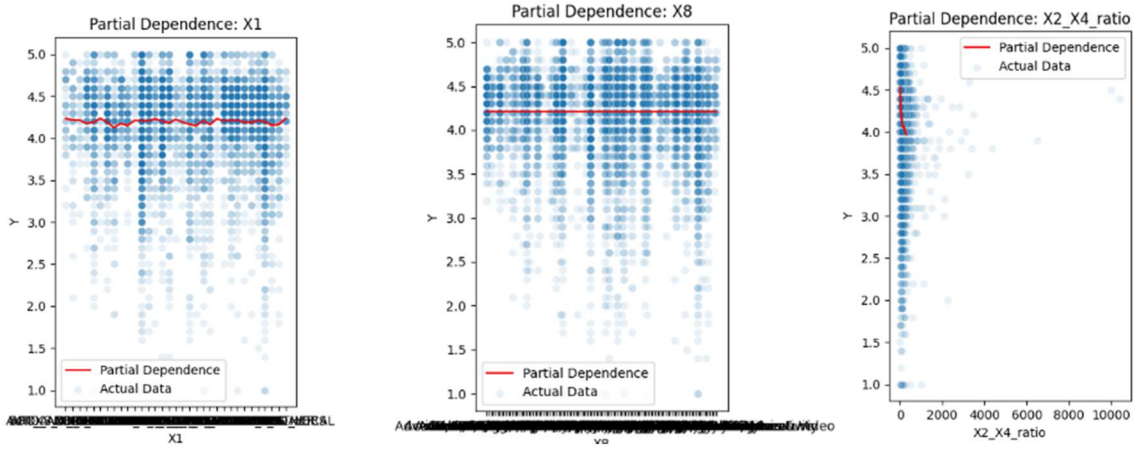
The idea behind choosing our features was to draw the correlation matrix and avoiding any pair of features that have a high correlation. Such features are called to have high multicollinearity. In regression models, we try to avoid features that have a higher correlation with each other than they have with Y, which we are aiming to predict.

So, these features were the perfect candidates for our model.



These graphs show the relationship between each of our features with our predicted rating Y.

Other features were tested to see their performance, other promising candidates were X[1] and X[8], since catboost shines with categorial features, these two were promising to use. Additionally, an engineered feature X2_X4_ratio was tested, along with other features, but they didn't give the overall best results.



4.0 Proposed approaches

An issue we encountered when inspecting our submission data, was that all values were heavily biased towards higher ratings (all Y values fall within the range between 4.7<Y<4.1).

After digging into the issue, and after reviewing our training data, we noticed that our training data was clustered towards that range, with a very few training data for low rating apps.

The following figure shows the distribution of Y in our training data:

```
Percentage distribution of values in the 'Y' column:
Range
0-1         0.186841
1-2         0.600561
2-3         3.243027
3-3.5       5.698652
3.5-4      17.696517
4-4.1       7.687175
4.1-4.3    21.753637
4.3-4.5    23.101561
4.5-4.7    13.919658
4.7-4.9     3.349793
4.9-5       2.762578
```

As we can notice, we have only 4% of our data for ratings below 3, and around 60% for 4.7<Y<4.1.

This issue aroused an idea that this could force the model to predict higher ratings very well, but fail to predict low rating apps due to a very low training data.

There were a couple of proposed solutions:

1) Duplicating some of the low rating apps was the simplest idea. But it is not very effective.
2) Another idea is to apply weights to different Y intervals to account for the low training data for specific Y intervals and feeding the weights to the model.
3) A different approach is to modify the loss function for the model to penalize bad predictions for low rated apps.
4) The last approach was to apply (Synthetic Minority Over-sampling Technique), or SMOTE for short, which in simple terms, tries to artificially generate new examples of the minority class rather than simply duplicating existing ones.
   It operates by applying k-nearest algorithm on a random minority class, and creates a new sample within that class.