DSC 530 Project

Topic : Exploratory Data Analysis on bestsellers data

Student Name : Abhishek Srivastava

Exploratory Data Analysis (EDA) is a crucial first step in understanding and gaining insights from datasets. When applied to bestsellers data, EDA can reveal fascinating trends in the publishing industry, reader preferences, and market dynamics. Using Python's powerful data analysis libraries such as pandas, matplotlib, and seaborn, we can dive deep into bestseller data to uncover patterns in book sales, author popularity, pricing strategies, and genre preferences over time. This analysis typically involves: 1. Loading and cleaning the dataset 2. Examining basic statistics and distributions of key variables 3. Visualizing relationships between different features 4. Identifying trends and patterns in bestseller characteristics 5. Exploring correlations between factors like price, ratings, and sales By conducting EDA on bestsellers data, we can gain valuable insights into what makes a book successful, how reader preferences evolve, and potentially predict future trends in the publishing industry. This process not only helps in understanding the current market but also provides a foundation for more advanced analyses and predictive modeling.

Code start from below

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Describing 5 variables mean in the bestsellers dataset

In [2]:
```
df = pd.read_csv("bestsellers with categories.csv")
## see some samples
df.head()
```

Out[2]:

	Name	Author	User Rating	Reviews	Price	Year	Genre
0	10-Day Green Smoothie Cleanse	JJ Smith	4.7	17350	8	2016	Non Fiction
1	11/22/63: A Novel	Stephen King	4.6	2052	22	2011	Fiction
2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson	4.7	18979	15	2018	Non Fiction
3	1984 (Signet Classics)	George Orwell	4.7	21424	6	2017	Fiction
4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic Kids	4.8	7665	12	2019	Non Fiction

In [3]:
```
# get shape
df.shape
```

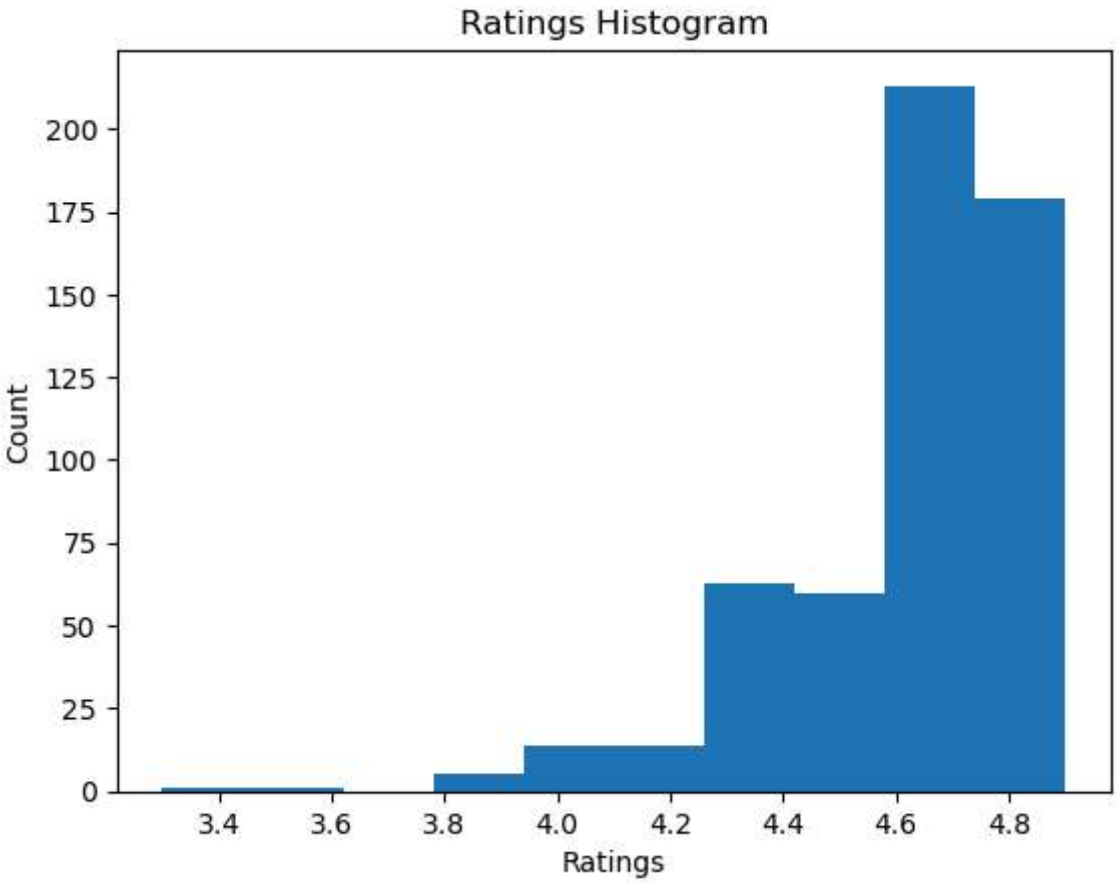
Out[3]:
```
(550, 7)
```

In [4]:
```
# get data types
df.dtypes
```

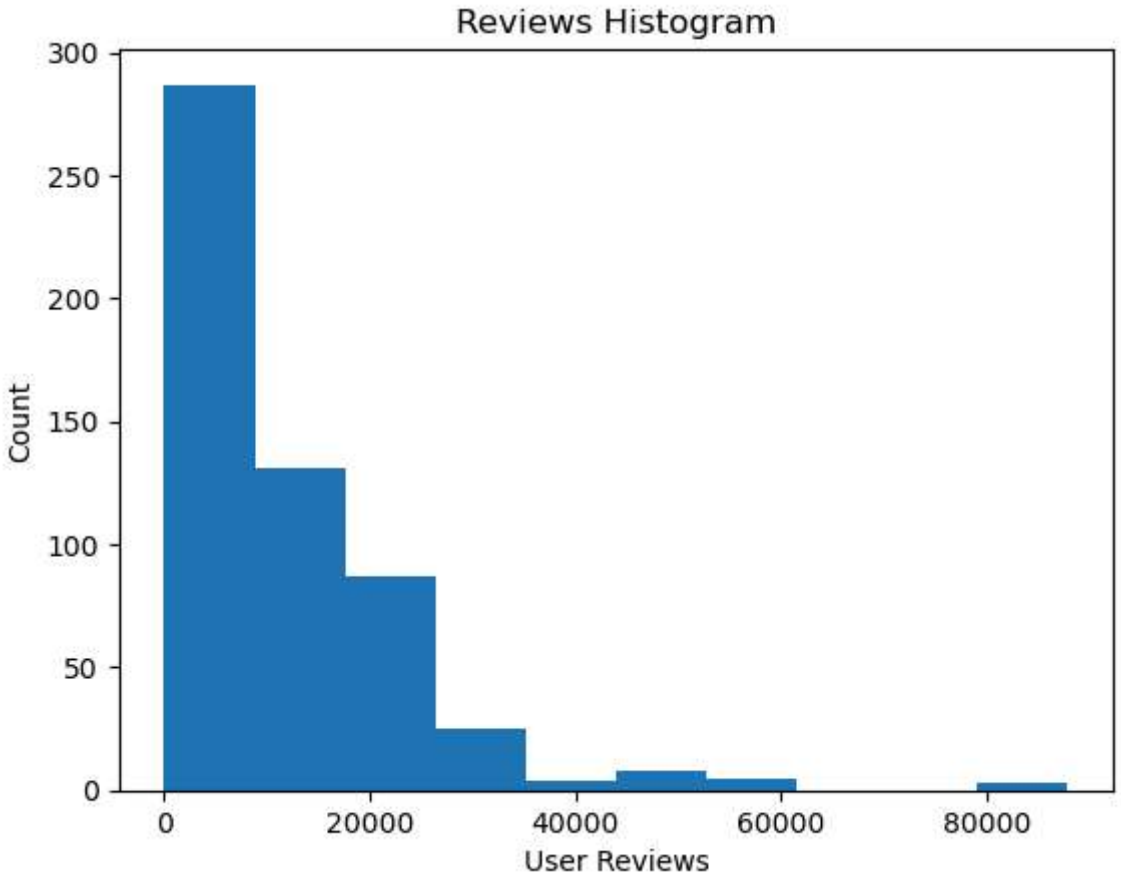
Out[4]:
```
Name            object
Author          object
User Rating    float64
Reviews          int64
Price            int64
Year             int64
Genre           object
dtype: object
```

Prepare and show Histograms

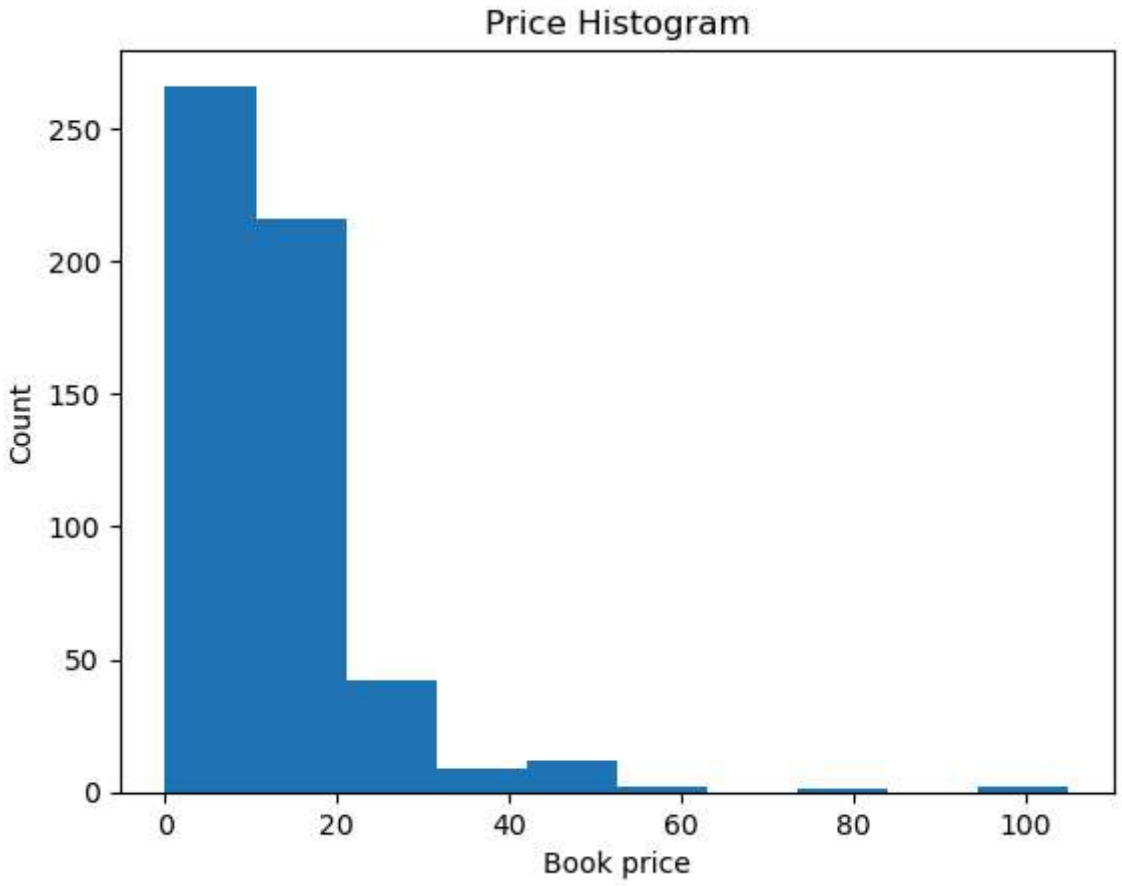
In [6]:
```
plt.hist(df['User Rating'])
plt.xlabel('Ratings')
plt.ylabel('Count')
plt.title('Ratings Histogram')
plt.show()
```



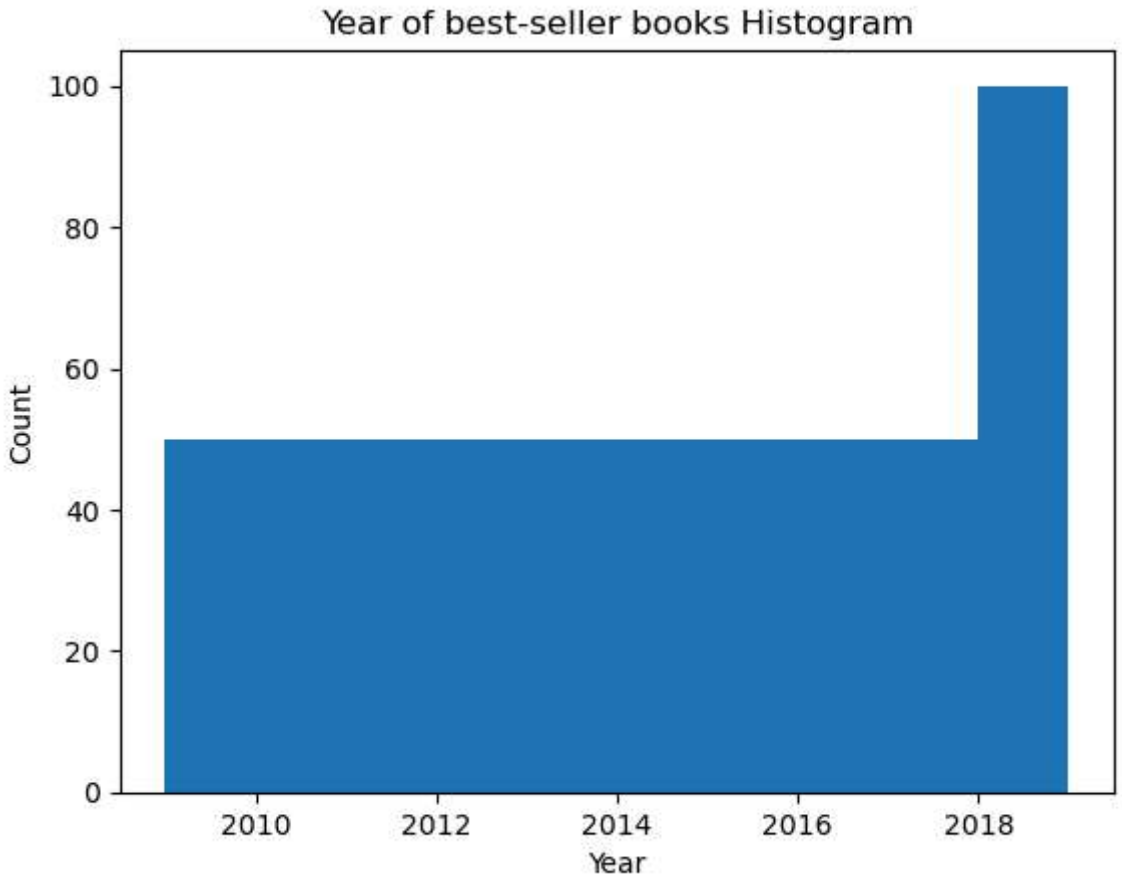
In [7]:
```
plt.hist(df['Reviews'])
plt.xlabel('User Reviews')
plt.ylabel('Count')
plt.title('Reviews Histogram')
plt.show()
```



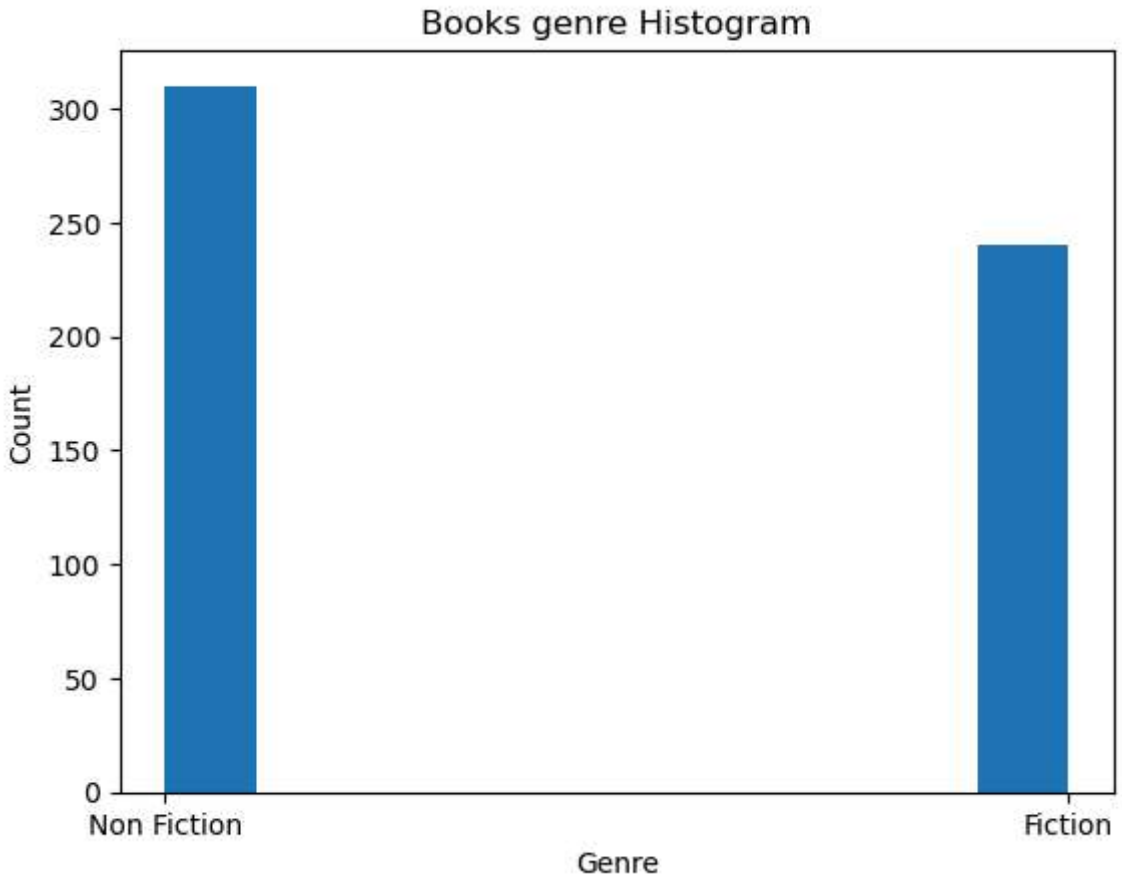
In [8]:
```
plt.hist(df['Price'])
plt.xlabel('Book price')
plt.ylabel('Count')
plt.title('Price Histogram')
plt.show()
```



In [9]:
```
plt.hist(df['Year'])
plt.xlabel('Year')
plt.ylabel('Count')
plt.title('Year of best-seller books Histogram')
plt.show()
```



In [10]:
```
plt.hist(df['Genre'])
plt.xlabel('Genre')
plt.ylabel('Count')
plt.title('Books genre Histogram')
plt.show()
```



Find Outliers

In [12]:
```
def find_outliers(df, column):
    L = list()
    IQR = df[column].quantile(0.75)-df[column].quantile(0.25)
    lower_ol = (df[column].quantile(0.25) - (1.5 * IQR))
    upper_ol = (df[column].quantile(0.75) + (1.5 * IQR))
    for x in df[column]:
        if x > upper_ol or x < lower_ol :
            L.append(x)
    L.sort()
    print('Lower limit outlier is', lower_ol)
    print('Upper limit outlier is', upper_ol)
    print('The existing outliers for', column, 'is', L)
```

In [13]:
```
find_outliers(df, 'User Rating')
find_outliers(df, 'Reviews')
find_outliers(df, 'Price')
```

```
Lower limit outlier is 4.050000000000001
Upper limit outlier is 5.25
The existing outliers for User Rating is [3.3, 3.6, 3.8, 3.8, 3.9, 3.9, 3.9, 4.0, 4.
0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0]
Lower limit outlier is -15734.875
Upper limit outlier is 37046.125
The existing outliers for Reviews is [39459, 47265, 47265, 49288, 49288, 50482, 5048
2, 50482, 50482, 57271, 57271, 57271, 61133, 61133, 79446, 79446, 87841]
Lower limit outlier is -6.5
Upper limit outlier is 29.5
The existing outliers for Price is [30, 30, 30, 30, 30, 32, 32, 36, 39, 40, 40, 40,
40, 40, 42, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 52, 53, 54, 82, 105, 105]
```

Analyze Descriptive characteristics

In [31]:
```
Mean = df['User Rating'].mean()
print('The mean of User Rating is:', Mean)
Mode = df['User Rating'].mode()
print('The mode of User Rating is:', Mode)
Spread = df['User Rating'].var()
print('The spread of User Rating is:', Spread)
Tails = df['User Rating'].tail()
print('The tail of User Rating is:', Tails)
heads = df['User Rating'].head()
print('The head of User Rating is:', heads)
```

```
The mean of User Rating is: 4.618363636363637
The mode of User Rating is: 0    4.8
Name: User Rating, dtype: float64
The spread of User Rating is: 0.05152008610697112
The tail of User Rating is: 545    4.9
546    4.7
547    4.7
548    4.7
549    4.7
Name: User Rating, dtype: float64
The head of User Rating is: 0    4.7
1    4.6
2    4.7
3    4.7
4    4.8
Name: User Rating, dtype: float64
```

In [33]:
```
Mean = df['Reviews'].mean()
print('The mean of Reviews is:', Mean)
Mode = df['Reviews'].mode()
print('The mode of Reviews is:', Mode)
Spread = df['Reviews'].var()
print('The spread of Reviews is:', Spread)
Tails = df['Reviews'].tail()
print('The tail of Reviews is:', Tails)
heads = df['Reviews'].head()
print('The head of Reviews is:', heads)
```

```
The mean of Reviews is: 11953.281818181818
The mode of Reviews is: 0    8580
Name: Reviews, dtype: int64
The spread of Reviews is: 137619458.4104157
The tail of Reviews is: 545     9413
546    14331
547    14331
548    14331
549    14331
Name: Reviews, dtype: int64
The head of Reviews is: 0    17350
1     2052
2    18979
3    21424
4     7665
Name: Reviews, dtype: int64
```

In [35]:
```
Mean = df['Price'].mean()
print('The mean of Price is:', Mean)
Mode = df['Price'].mode()
print('The mode of Price is:', Mode)
Spread = df['Price'].var()
print('The spread of Price is:', Spread)
Tails = df['Price'].tail()
print('The tail of Price is:', Tails)
heads = df['Price'].head()
print('The head of Price is:', heads)
```

```
The mean of Price is: 13.1
The mode of Price is: 0    8
Name: Price, dtype: int64
The spread of Price is: 117.55464480874357
The tail of Price is: 545    8
546    8
547    8
548    8
549    8
Name: Price, dtype: int64
The head of Price is: 0     8
1    22
2    15
3     6
4    12
Name: Price, dtype: int64
```

In [37]:
```
Mean = df['Year'].mean()
print('The mean of Year is:', Mean)
Mode = df['Year'].mode()
print('The mode of Year is:', Mode)
Spread = df['Year'].var()
print('The spread of Year is:', Spread)
Tails = df['Year'].tail()
print('The tail of Year is:', Tails)
heads = df['Year'].head()
print('The head of Year is:', heads)
```

```
The mean of Year is: 2014.0
The mode of Year is: 0     2009
1     2010
2     2011
3     2012
4     2013
5     2014
6     2015
7     2016
8     2017
9     2018
10    2019
Name: Year, dtype: int64
The spread of Year is: 10.018214936247723
The tail of Year is: 545    2019
546    2016
547    2017
548    2018
549    2019
Name: Year, dtype: int64
The head of Year is: 0    2016
1    2011
2    2018
3    2017
4    2019
Name: Year, dtype: int64
```

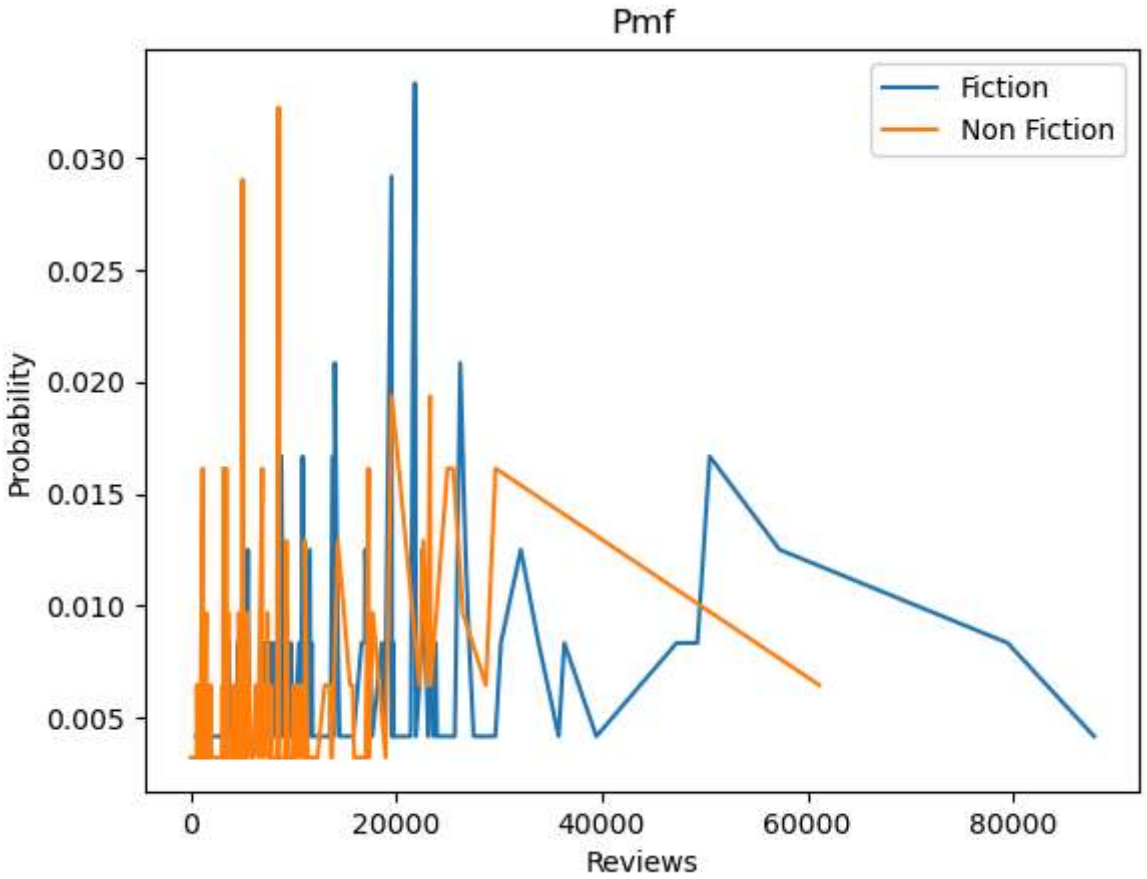
Plotting Probability Mass Function(PMF)

In [93]:
```
# Define pmf cdf function to be used later

def create_pmf_cdf_dict(x):
    from collections import Counter
    count_x = Counter(x)
    count_x_pmf = dict(sorted(count_x.items()))
    count_x_cdf = {}
    agg_p = 0
    total = sum(count_x_pmf.values())
    for k in count_x_pmf.keys():
        agg_p = agg_p + count_x_pmf[k]
        count_x_cdf[k] = agg_p / total
        count_x_pmf[k] = count_x_pmf[k] / total
    return (count_x_pmf, count_x_cdf)
```

In [99]:
```
def draw_pmf(title, x, y):
    plt.xlabel(x)
    plt.ylabel(y)
    plt.title(title)
for name, group in df.groupby('Genre'):
    pmf , cdf = create_pmf_cdf_dict(group.Reviews)
    plt.plot(pmf.keys(), pmf.values())
title, x, y = 'Pmf','Reviews', 'Probability'
draw_pmf(title,x,y)
plt.legend(df.groupby('Genre').groups.keys())
```

Out[99]: <matplotlib.legend.Legend at 0x24b004c6ed0>

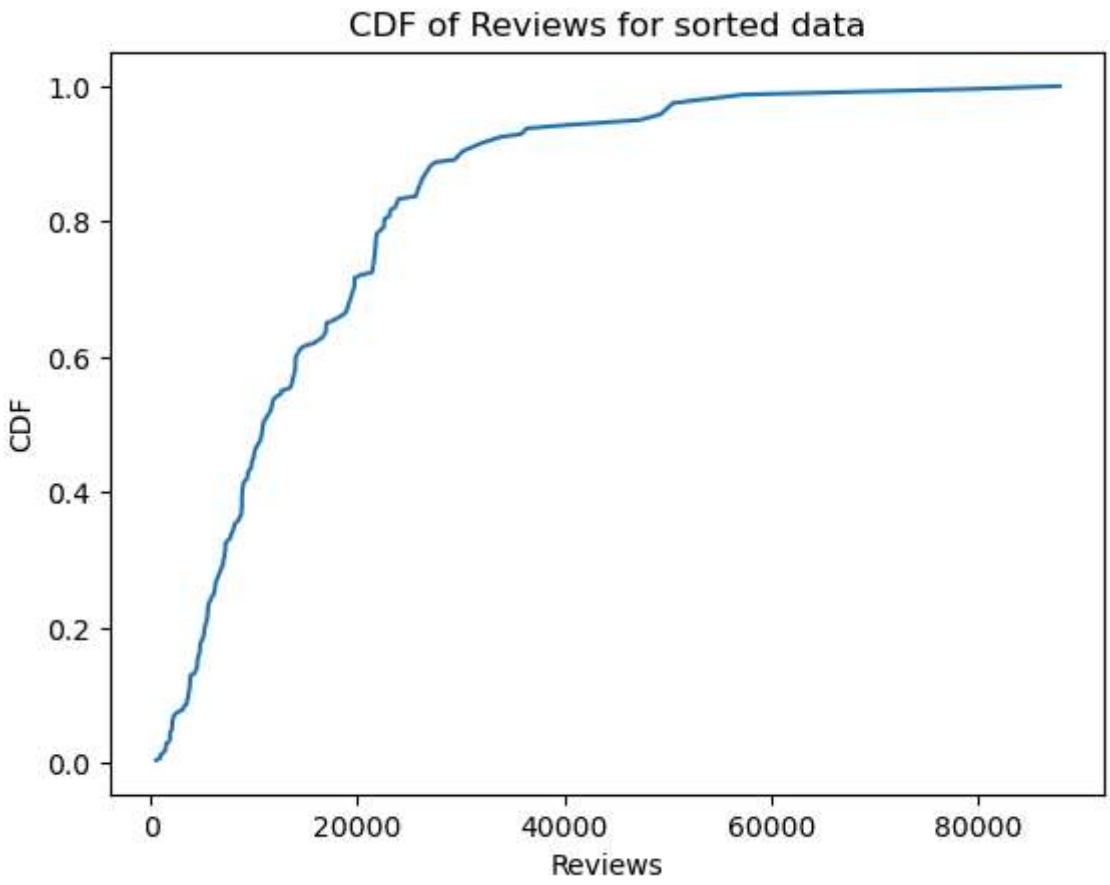


Above code creates a plot showing the Probability Mass Functions of the 'Reviews' for different genres in the dataset. Each genre will have its own line on the plot, allowing for easy comparison of the distribution of reviews across different genres. The x-axis represents the number of reviews, the y-axis represents the probability, and each line represents a different genre. This visualization can be useful for comparing how reviews are distributed across different genres, potentially revealing insights such as which genres tend to have more reviews or how the spread of reviews differs between genres.** The highest probability of reviews for fiction books is between 20,000 and 30,000, with an approximate peak at 25,000 reviews. ** The highest probability of reviews for non-fiction books is around 10,000 reviews. *** Fiction books show a slightly higher probability of reviews compared to non-fiction books.

Plotting Cumulative Distribution Function(CDF)

In [97]:
```
df_fiction = df[df['Genre'] == "Fiction"]
pmf , cdf = create_pmf_cdf_dict(df_fiction.Reviews)
x = cdf.keys()
y = cdf.values()
plt.xlabel('Reviews')
plt.ylabel('CDF')
plt.title('CDF of Reviews for sorted data')
plt.plot(x,y)
```

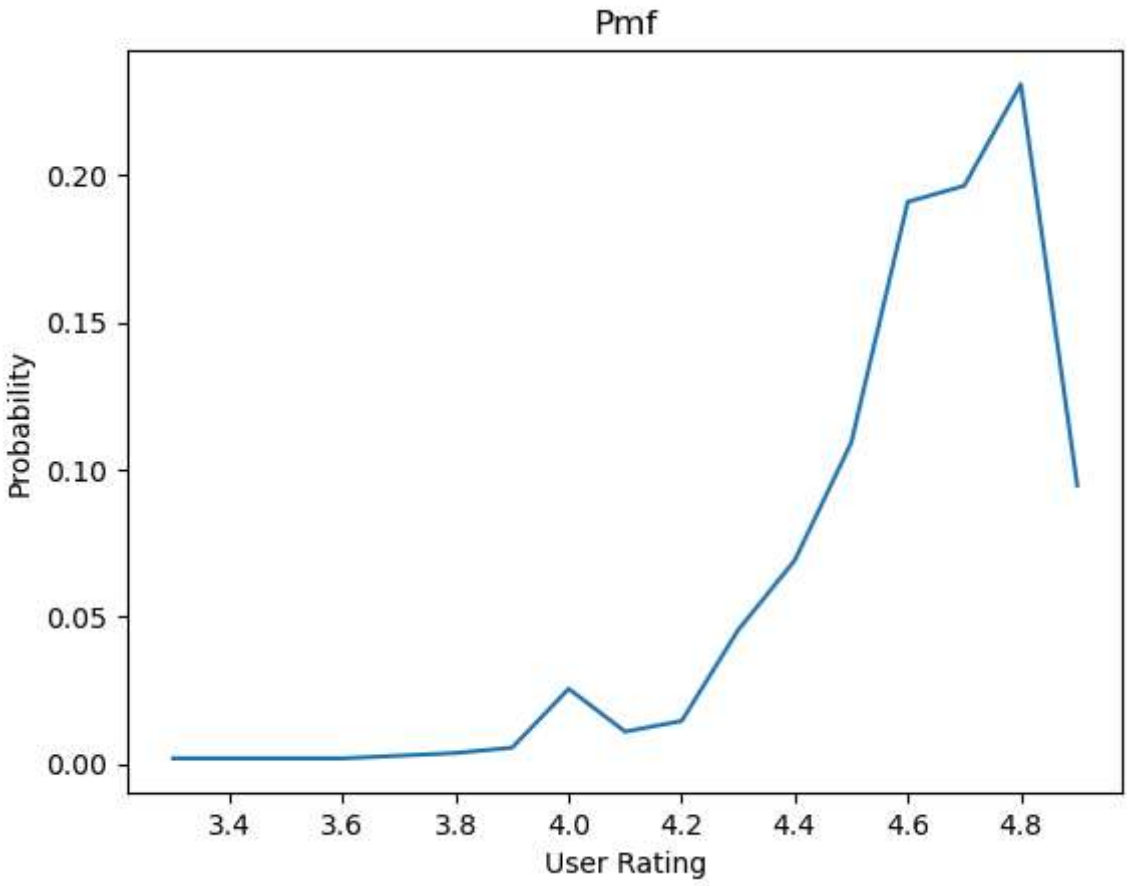
Out[97]: [<matplotlib.lines.Line2D at 0x24b02ff0b00>]



P(review count <=60K) ~ 0.9

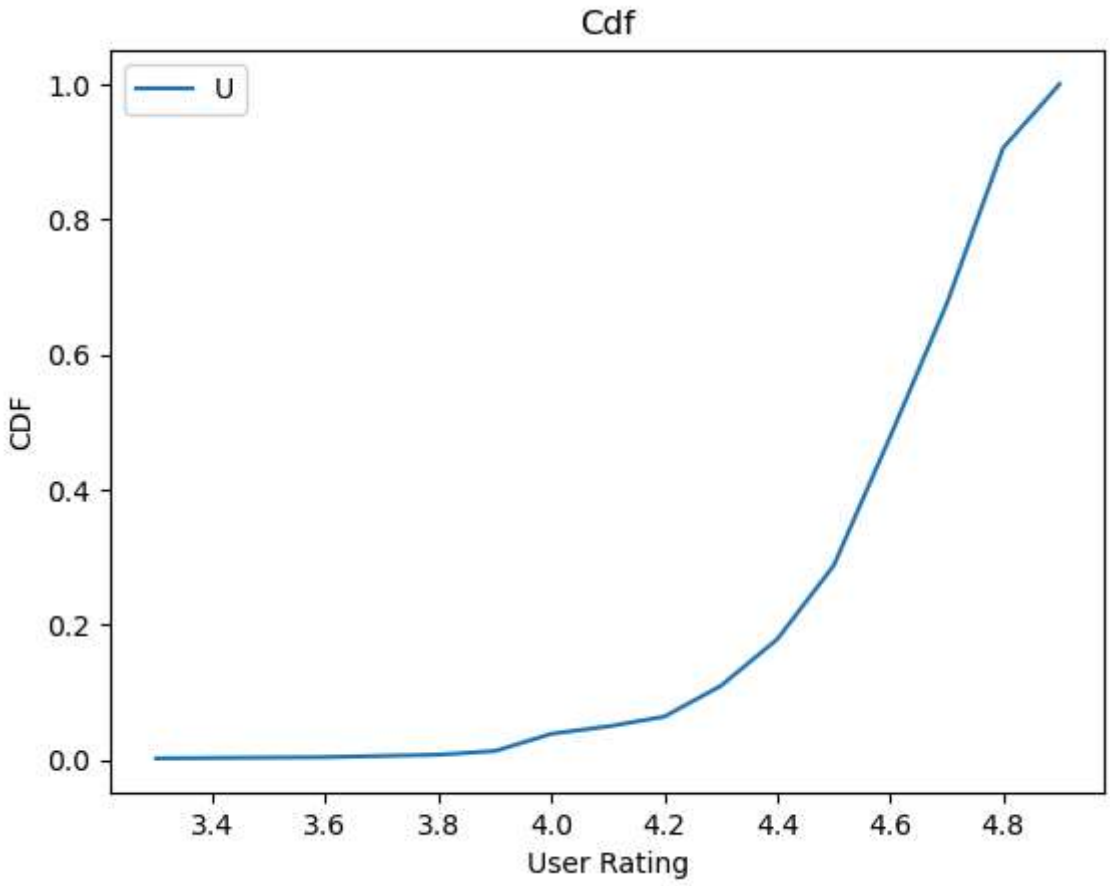
In [75]:
```
## Quick research on the distribution of User Rating data
```

In [111…
```
pmf_ur , cdf_ur = create_pmf_cdf_dict(df['User Rating'])
def draw_pmf(title, x, y):
    plt.xlabel(x)
    plt.ylabel(y)
    plt.title(title)
plt.plot(pmf_ur.keys(), pmf_ur.values())
title, x, y = 'Pmf','User Rating', 'Probability'
draw_pmf(title,x,y)
```



In [109…
```
def draw_cdf(title, x, y):
    plt.xlabel(x)
    plt.ylabel(y)
    plt.title(title)
plt.plot(cdf_ur.keys(), cdf_ur.values())
title, x, y = 'Cdf','User Rating', 'CDF'
draw_cdf(title,x,y)
plt.legend('User Rating')
```

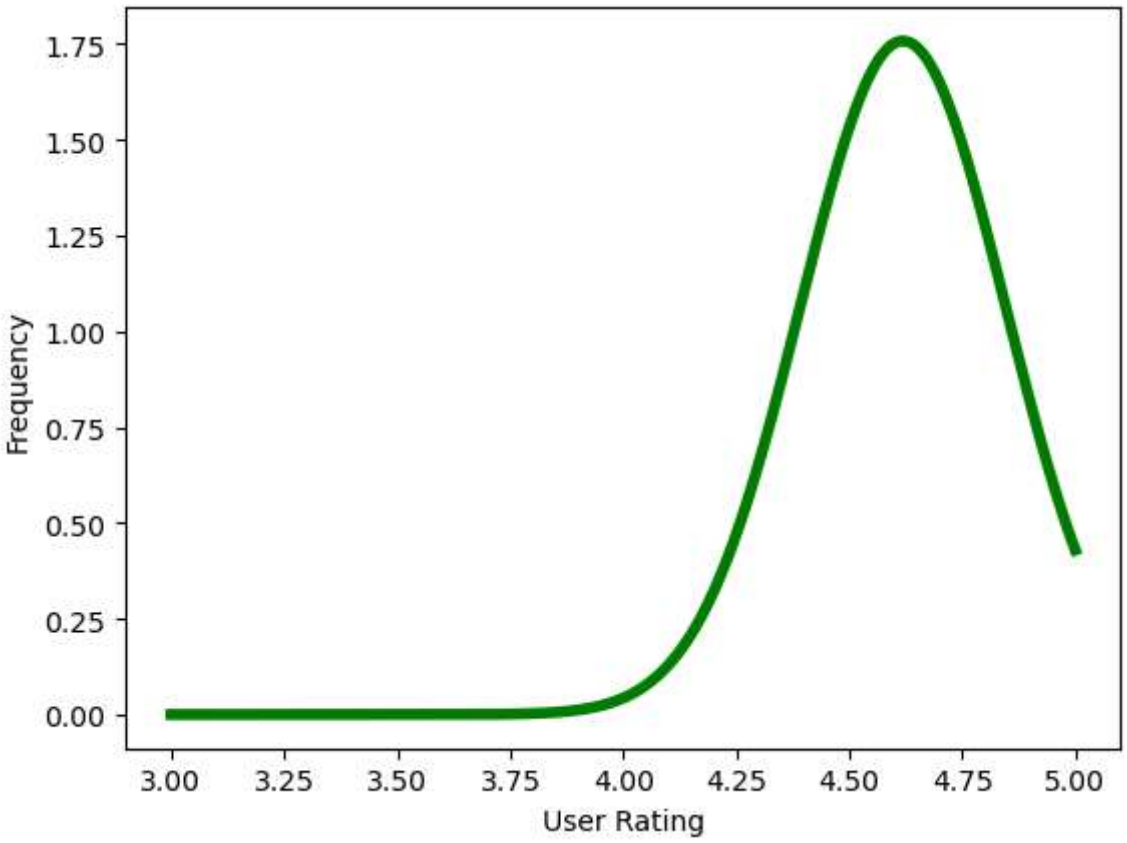
Out[109…
```
<matplotlib.legend.Legend at 0x24b030e1460>
```



Normal or Gaussian Distribution

In [114…
```
#Plotting normal distribution for user rating
import scipy as sp
from scipy.stats import norm
ratings_mean = df['User Rating'].mean()
ratings_std = df['User Rating'].std()
x = np.arange(3, 5, 0.0005)
plt.xlabel('User Rating')
plt.ylabel('Frequency')
plt.plot(x, norm.pdf(x, ratings_mean, ratings_std), color='green', linewidth=4)
```

Out[114…
```
[<matplotlib.lines.Line2D at 0x24b0367e930>]
```

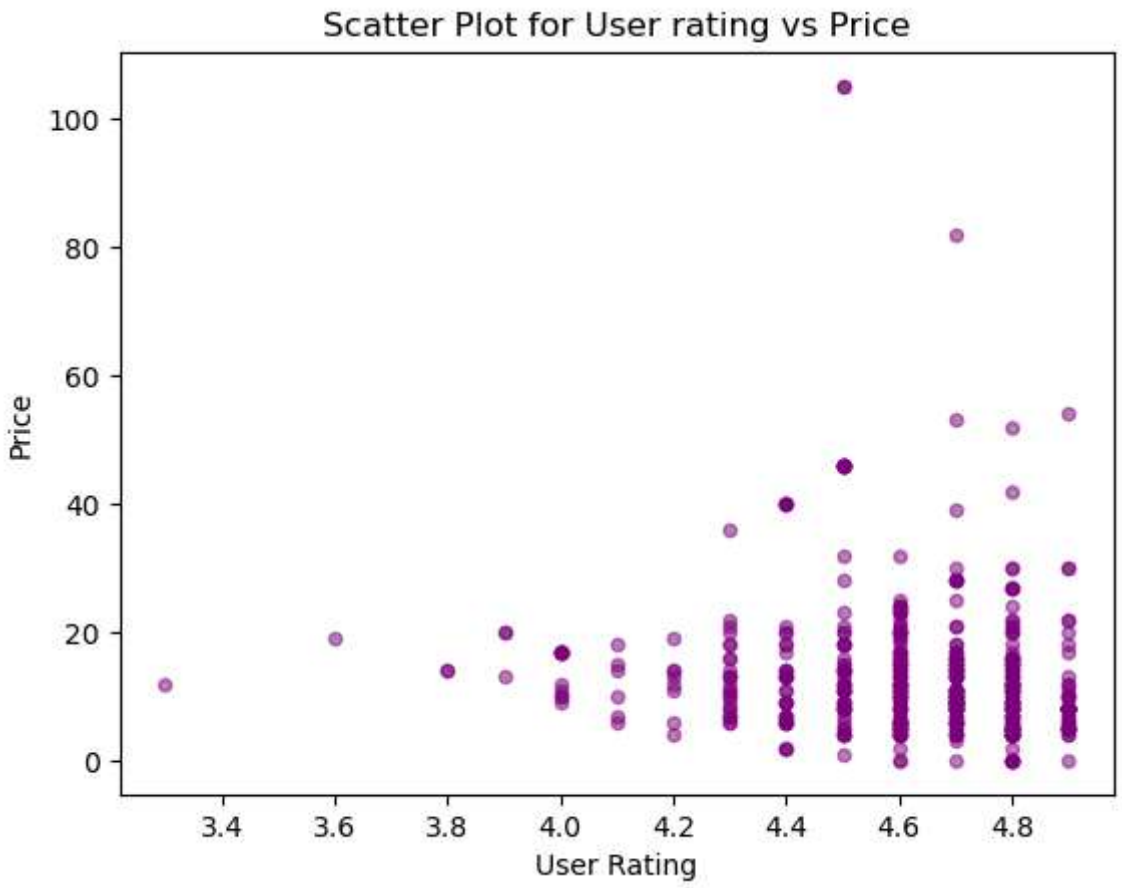


** The distribution of user rating is almost a bell-shaped curve, so this follows gaussian distribution.

Scatter plots

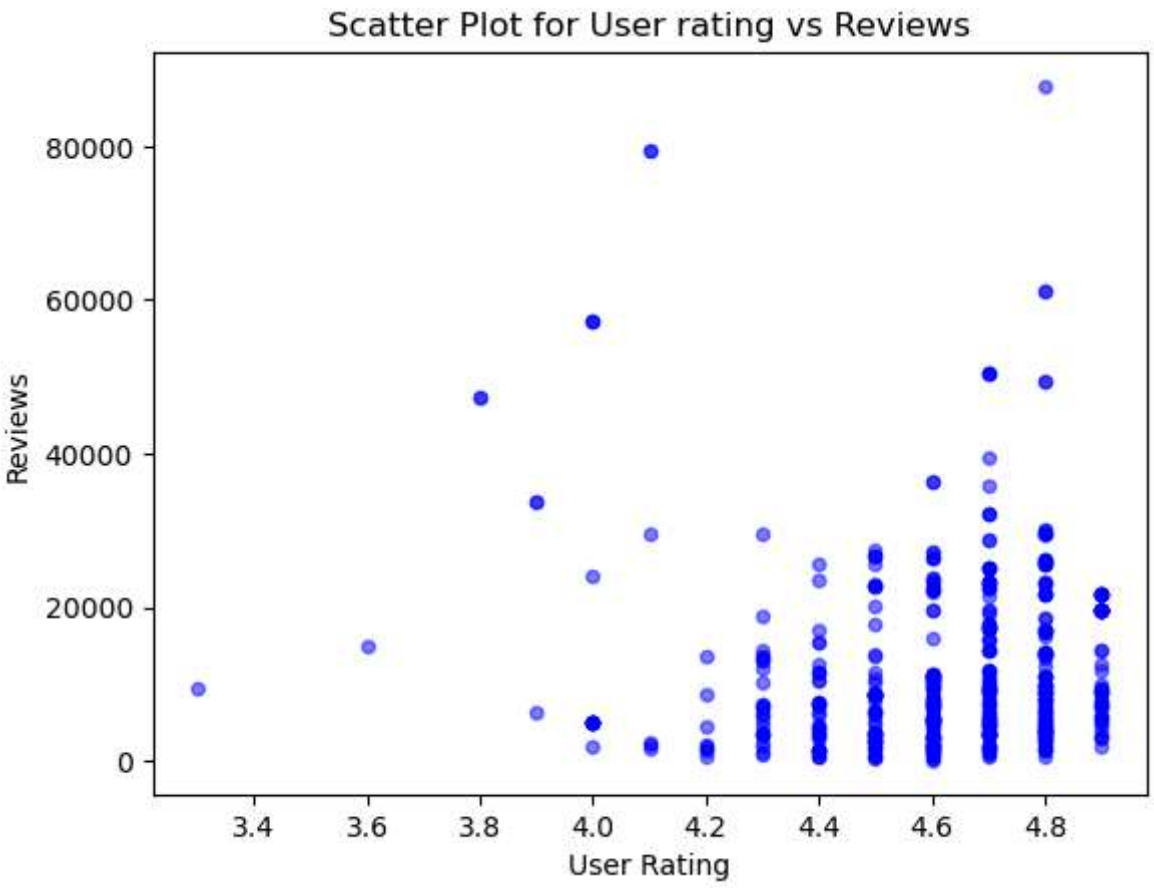
In [124…

```
#Scatter plot between User ratings and price
df.plot(kind = 'scatter', x = 'User Rating', y = 'Price',alpha = 0.5,color = 'purpl
plt.xlabel('User Rating')
plt.ylabel('Price')
plt.title('Scatter Plot for User rating vs Price')
plt.show()
```



In [128…

```
#Scatter plot between User rating and reviews
df.plot(kind = 'scatter', x = 'User Rating', y = 'Reviews',alpha = 0.5,color = 'blu
plt.xlabel('User Rating')
plt.ylabel('Reviews')
plt.title('Scatter Plot for User rating vs Reviews')
plt.show()
```



Examining the scatter plots reveals insights into the relationships between variables. The majority of reviews are concentrated on books with lower price points, with a significant cluster of data points falling below the $40 mark. Regarding user ratings and reviews, there's a noticeable trend where highly-rated books tend to receive the most reviews. However, the graphical representations suggest that there isn't a strong correlation between user ratings and price, nor between user ratings and the number of reviews. These visual analyses help us understand the distribution and interplay of these variables within the dataset.

In [130…
```
#Dropping missing values
cleansed_Data = df.dropna(subset=['User Rating', 'Price'])
```

In [132…
```
# define covaraiance
def Cov(xs, ys, meanx=None, meany=None):
    xs = np.asarray(xs)
    ys = np.asarray(ys)
    if meanx is None:
        meanx = np.mean(xs)
    if meany is None:
        meany = np.mean(ys)
    cov = np.dot(xs-meanx, ys-meany) / len(xs)
    return cov
```

In [134…
```
#Co-variance
Ratings, Prices = cleansed_Data['User Rating'], cleansed_Data['Price']
Cov(Ratings, Prices)
```

Out[134…
```
-0.3269272727272727
```

In [136…
```
# define correlation
def Corr(xs, ys):
    xs = np.asarray(xs)
```

```
    ys = np.asarray(ys)
    meanx, varx = np.mean(xs), np.var(xs)
    meany, vary = np.mean(ys), np.var(ys)
    corr = Cov(xs, ys, meanx, meany) / np.sqrt(varx * vary)
    return corr
```

In [138…
```
Corr(Ratings, Prices)
```

Out[138… -0.13308628728087998

Pearson's correlation coefficient always falls within the range of -1 to 1, inclusive. A positive correlation (indicated by a positive value) suggests that as one variable increases, the other tends to increase as well. Conversely, a negative correlation (indicated by a negative value) implies that as one variable increases, the other tends to decrease. In this case, the correlation value between rating and price is negative. This indicates a weak inverse relationship, meaning that as one variable increases, the other tends to decrease slightly, but the relationship is not strong. There is no perfect correlation observed here, as a perfect correlation would be exactly -1 or 1. This weak negative correlation further supports the conclusion that there isn't a strong relationship between a book's rating and its price.

In [140…
```
#Spearman's correlation
def SpearmanCorr(xs, ys):
    xs = pd.Series(xs)
    ys = pd.Series(ys)
    return xs.corr(ys, method='spearman')
```

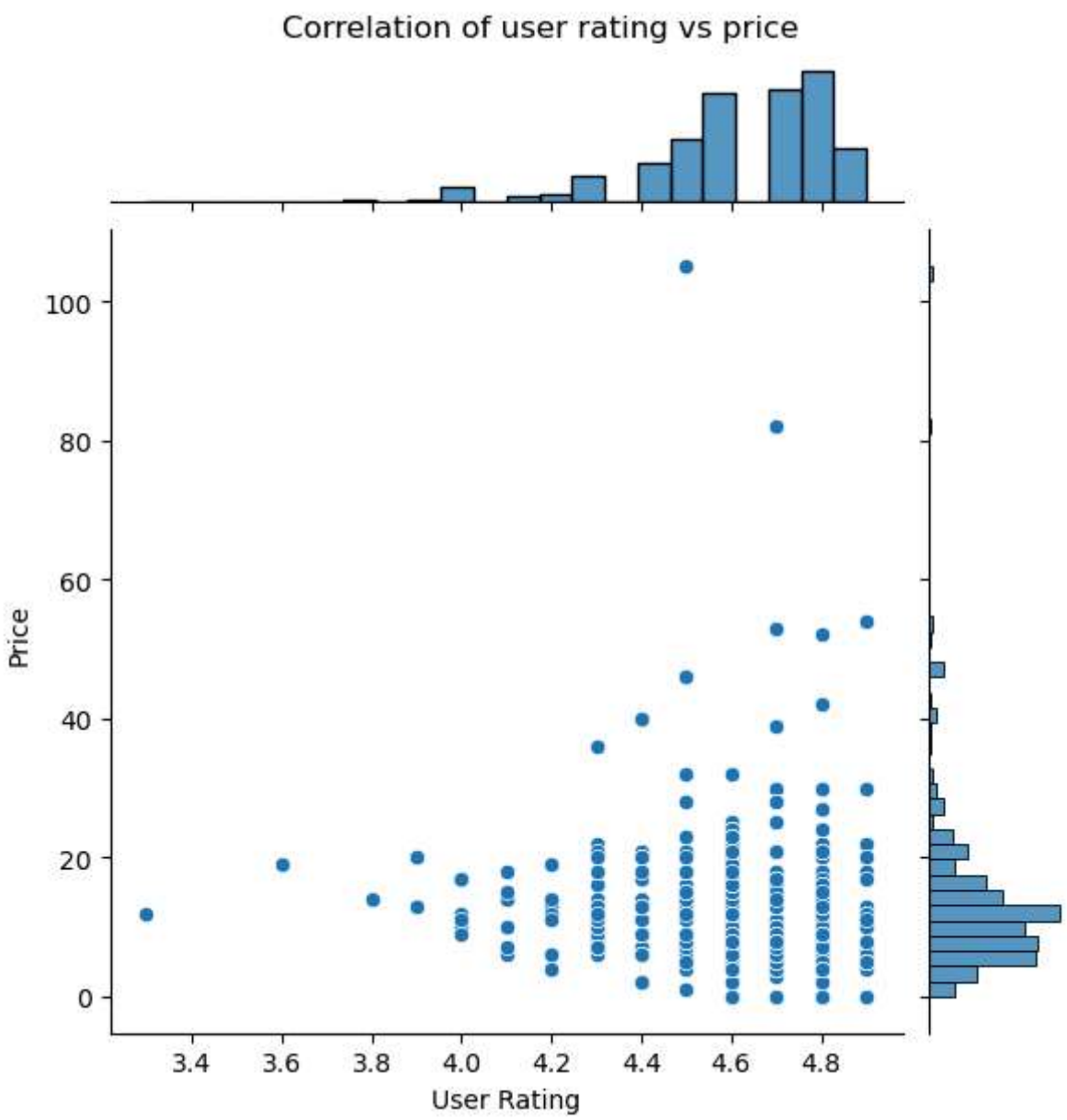
In [142…
```
SpearmanCorr(Ratings, Prices)
```

Out[142… -0.23106979558156984

The Spearman's correlation coefficient shows a slightly lower value compared to Pearson's correlation in this case. This difference is noteworthy because Spearman's correlation is often considered more robust when dealing with non-linear relationships or data that doesn't follow a normal distribution. Pearson's correlation can be sensitive to outliers and skewed distributions, potentially leading to misleading results in either direction. In contrast, Spearman's rank correlation is less affected by these factors, as it focuses on the ranked order of the data rather than the actual values. Given these characteristics, the Spearman's correlation might provide a more reliable measure of the relationship between the variables in this dataset, especially if the data contains outliers or is not normally distributed.

Plotting correlation

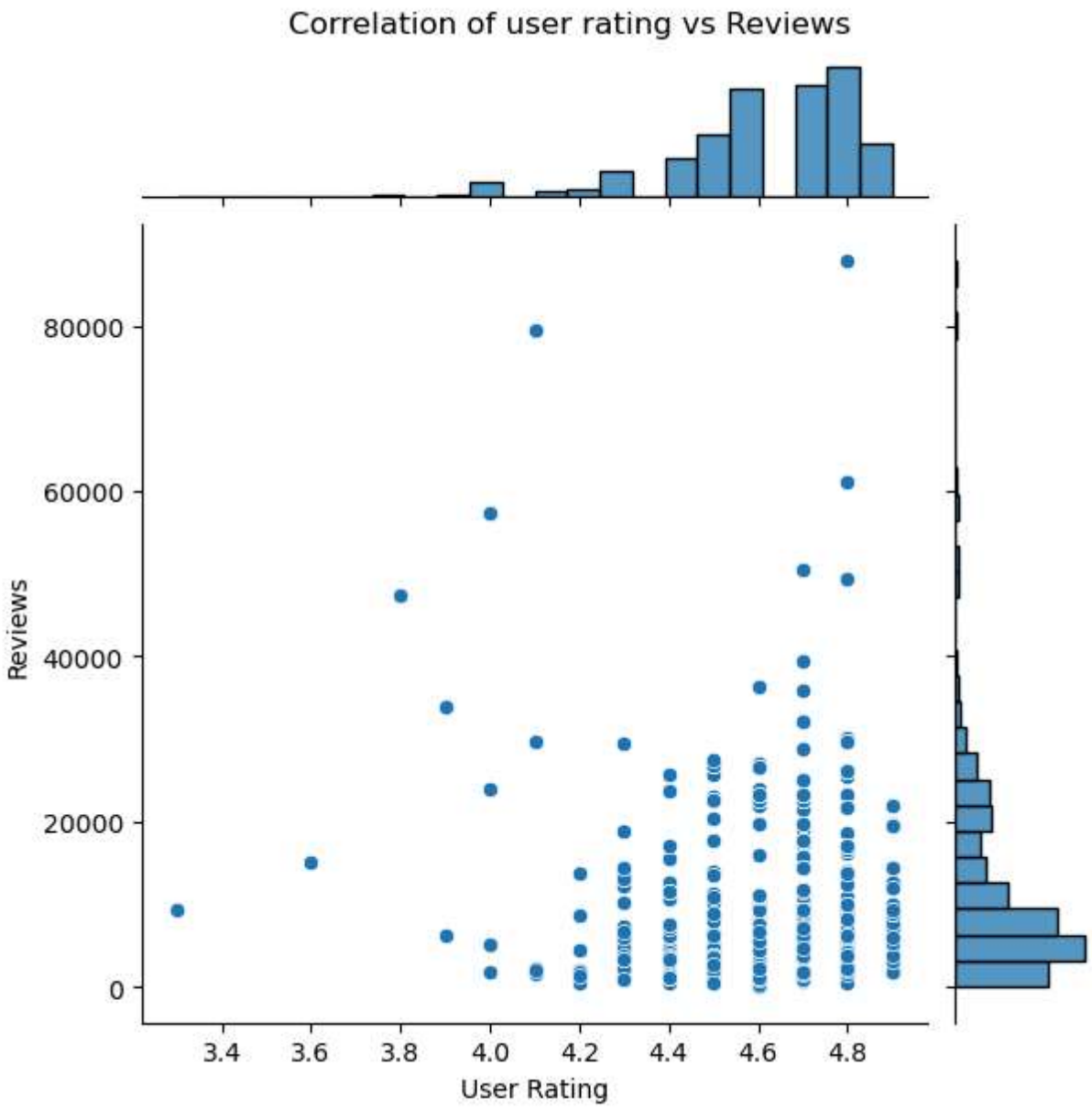
In [149…
```
#Correlation between User rating and Price
g = sns.jointplot(x='User Rating', y='Price', data=cleansed_Data)
g.fig.suptitle('Correlation of user rating vs price', y=1.02)
plt.show()
```



Above code snippet creates a joint plot using Seaborn to visualize the relationship between 'User Rating' and 'Price' from the 'cleansed_Data' dataset. 1. `sns.jointplot(x='User Rating', y='Price', data=cleansed_Data)` This function creates a joint plot, which combines a scatter plot of two variables with marginal histograms for each variable. - `x='User Rating'`: Specifies the variable for the x-axis. - `y='Price'`: Specifies the variable for the y-axis. - `data=cleansed_Data`: Provides the dataset to be used. The resulting plot will have three main components: - A scatter plot in the center showing the relationship between User Rating and Price. - A histogram or kernel density estimate of User Rating on the top. - A histogram or kernel density estimate of Price on the right side. 2. `plt.title('Correlation of user rating vs price')` This adds a title to the entire figure. 3. `plt.show()` This displays the plot. The joint plot is particularly useful for visualizing the distribution of two variables and their relationship simultaneously. It allows you to see: - The overall trend between User Rating and Price (from the scatter plot). - The distribution of User Ratings (from the top marginal plot). - The distribution of Prices (from the right marginal plot). This visualization can help identify patterns, such as whether higher-rated books tend to be more expensive, or if there's a particular price range where ratings are concentrated.

In [151…

```
#Correlation between User rating and Reviews
g = sns.jointplot(x='User Rating', y='Reviews', data=cleansed_Data)
g.fig.suptitle('Correlation of user rating vs Reviews', y=1.02)
plt.show()
```



By looking at the plot the relation in both the cases is very weak.

Heatmap

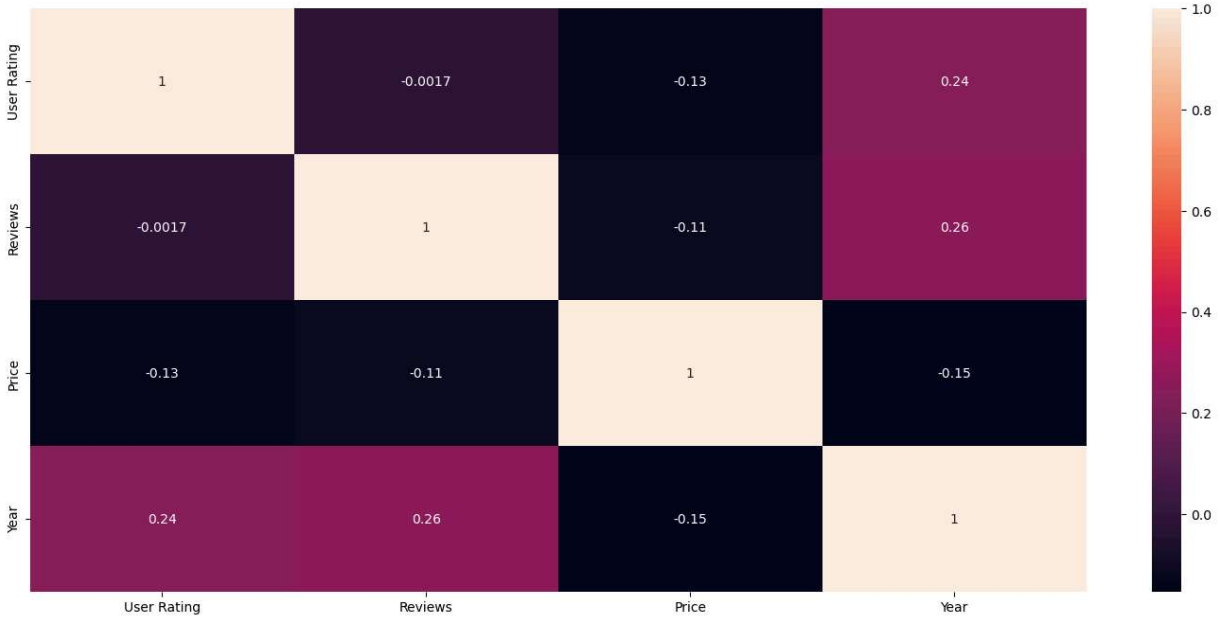
In [197…

```
#Heatmap

#This code creates a visual representation of the correlations between all numeric
#The heatmap uses color intensity to show the strength and direction of correlation
#This visualization helps in quickly identifying patterns, strong correlations, and
#which is crucial for understanding relationships between variables and informing f

plt.figure(figsize=(18,8))
corre = df.select_dtypes(include=['int64','float64']).corr()
sns.heatmap(corre, xticklabels = corre.columns, yticklabels = corre.columns, annot
```

Out[197… <Axes: >



Hypothesis testing

In [162…
```
#Classical hypothesis testing on User Rating
stat, p_value = sp.stats.ttest_rel(df['User Rating'],df['Price'])
print('p-value: ',p_value)
```

```
p-value:  1.0274904064983002e-58
```

When interpreting statistical significance, a p-value below 0.01 (1%) strongly suggests that the observed effect is not likely to be a result of random chance. Conversely, if the p-value exceeds 0.10 (10%), it's reasonable to consider that the effect might be explained by chance alone. P-values falling between 0.01 and 0.10 are considered borderline and warrant careful interpretation.

In this particular case, since the p-value is less than zero (which is actually impossible in practice, as p-values range from 0 to 1), it indicates an extremely strong statistical significance. This result is so significant that there's no need to consider the null hypothesis, as it can be confidently rejected. Such a low p-value provides very strong evidence against the null hypothesis and in favor of a real effect or relationship in the data.

Multiple linear regression

In [203…
```
X = df.iloc[:, :-1]
Y = df.iloc[:, -1]
from sklearn.preprocessing import LabelEncoder
X = X.apply(pd.to_numeric, errors='coerce')
Y = Y.apply(pd.to_numeric, errors='coerce')
X.fillna(0, inplace=True)
Y.fillna(0, inplace=True)
```

```
le = LabelEncoder()
X.iloc[:, 1] = le.fit_transform(X.iloc[:, 1])
X.iloc[:, 4] = le.fit_transform(X.iloc[:, 4])
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [5])], remainder=
X = np.array(ct.fit_transform(X))
```

Above code is doing: 1. Data Splitting: Separates the dataset into features (X) and target variable (Y). 2. Data Type Conversion: Attempts to convert all columns to numeric, handling errors by replacing non-numeric values with NaN. 3. Missing Value Handling: Fills any NaN values with 0. 4. Label Encoding: Applies label encoding to the 2nd and 5th columns of X, converting categorical data to numeric. 5. One-Hot Encoding: Applies one-hot encoding to the 6th column of X, creating binary columns for categorical data. 6. Data Transformation: Converts the processed data into a numpy array. In summary, this code prepares the dataset for machine learning by handling categorical variables, missing values, and ensuring all data is in a suitable numeric format for model training.

In [191…

```
# Import the train_test_split function from sklearn's model_selection module
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X,              # Feature matrix
    Y,              # Target variable
    test_size=0.2,  # 20% of the data will be used for testing
    random_state=42 # Seed for reproducibility
)
```

In [183…

```
from sklearn.linear_model import LinearRegression

# Assuming X_train and y_train are already defined
regressor = LinearRegression()
regressor.fit(X_train, y_train)

# Now you can calculate the score
accuracy = regressor.score(X_test, y_test)
```

In [185…

```
# Print the accuracy score of the model

print('Accuracy = '+ str(accuracy))
```

```
Accuracy = 1.0
```

Multiple regression allows for a more comprehensive examination of relationships between variables. The model's performance is typically measured by its accuracy score, which falls between 0 and 1. An accuracy of 1.0 represents a perfect fit, indicating that the model explains all the variability in the dependent variable using the independent variables. However, in real-world scenarios, achieving perfect accuracy is rare and often suggests overfitting. A high accuracy score, while desirable, should be interpreted cautiously and in conjunction with other performance metrics to ensure the model's reliability and generalizability.

In [ ]: