In [518…
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

np.random.seed(42)  # for reproducibility
```

In [519…
```
df = pd.read_csv("amazon_updated.csv")

df.head()
```

Out[519…

	product_id	product_name	category	disc
0	B07JW9H4J1	Wayona Nylon Braided USB to Lightning Fast Cha...	Computers&Accessories|Accessories&Peripherals|...	
1	B098NS6PVG	Ambrane Unbreakable 60W / 3A Fast Charging 1.5...	Computers&Accessories|Accessories&Peripherals|...	
2	B096MSW6CT	Sounce Fast Phone Charging Cable & Data Sync U...	Computers&Accessories|Accessories&Peripherals|...	
3	B08HDJ86NZ	boAt Deuce USB 300 2 in 1 Type-C & Micro USB S...	Computers&Accessories|Accessories&Peripherals|...	
4	B08CF3B7N1	Portronics Konnect L 1.2M Fast Charging 3A 8 P...	Computers&Accessories|Accessories&Peripherals|...	

In [520…
```
df.describe()
```

Out[520…

	rating
count	1464.000000
mean	4.096585
std	0.291674
min	2.000000
25%	4.000000
50%	4.100000
75%	4.300000
max	5.000000

In [521…

```
df = df[['discounted_price', 'actual_price',
         'discount_percentage', 'rating', 'rating_count']]
df.head()
```

Out[521…

	discounted_price	actual_price	discount_percentage	rating	rating_count
0	399	1,099	64%	4.2	24,269
1	199	349	43%	4.0	43,994
2	199	1,899	63%	3.9	7,928
3	329	699	53%	4.2	94,363
4	154	399	61%	4.2	16,905

In [522…

```
# Check for null values in the specified columns
null_check = {
    'rating': df['rating'].isnull().sum(),
    'rating_count': df['rating_count'].isnull().sum(),
    'discounted_price': df['discounted_price'].isnull().sum(),
    'actual_price': df['actual_price'].isnull().sum(),
    'discount_percentage': df['discount_percentage'].isnull().sum()
}

print("Number of null values in each column:")
for col, count in null_check.items():
    print(f"{col}: {count}")
```

```
Number of null values in each column:
rating: 0
rating_count: 2
discounted_price: 0
actual_price: 0
discount_percentage: 0
```

In [523…

```
# Drop rows with null values in rating and rating_count columns
df = df.dropna(subset=['rating_count'])

print(f"DataFrame shape after dropping null values: {df.shape}")
```

```
print("\nNull values remaining:")

print(df[['rating_count']].isnull().sum())
```

```
DataFrame shape after dropping null values: (1462, 5)

Null values remaining:
rating_count    0
dtype: int64
```

In [524…

```
# Data Cleaning

df['rating'] = df['rating'].pipe(pd.to_numeric, errors='coerce')
df['rating_count'] = df['rating_count'].astype(str).str.replace(',', '').pip
df['discount_percentage'] = df['discount_percentage'].str.replace("%", "").a

df['discounted_price'] = df['discounted_price'].astype(str).str.replace(',',
df['actual_price'] = df['actual_price'].astype(
    str).str.replace(',', '').pipe(pd.to_numeric, errors='coerce')

df.head()
```

Out[524…

	discounted_price	actual_price	discount_percentage	rating	rating_count
0	399.0	1099.0	64	4.2	24269
1	199.0	349.0	43	4.0	43994
2	199.0	1899.0	63	3.9	7928
3	329.0	699.0	53	4.2	94363
4	154.0	399.0	61	4.2	16905

In [525…

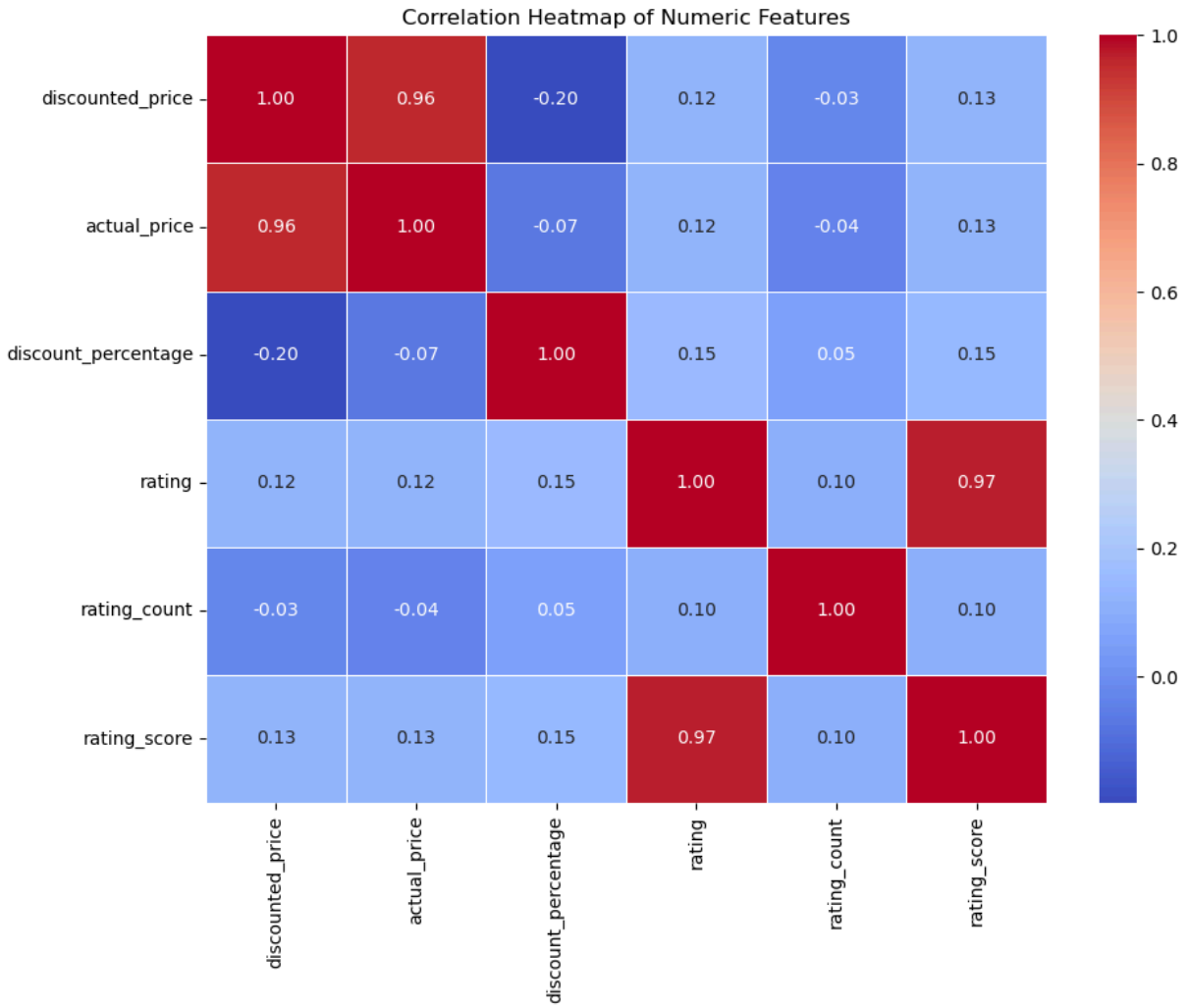
```
#Feature Engineering
df = df.dropna()


mean_rating = df['rating'].mean()
min_ratings = 15
df['rating_score'] = (df['rating_count'] * df['rating'] +
                      min_ratings * mean_rating) / (df['rating_count'] + mir
if df['rating_score'].isnull().any():
    df['rating_score'].fillna(df['rating_score'].median(), inplace=True)

df.describe()
```

Out [525…

	discounted_price	actual_price	discount_percentage	rating	rating_
count	1462.000000	1462.000000	1462.000000	1462.000000	1462.0
mean	3129.981826	5453.087743	45.216826	4.096717	18307.3
std	6950.548042	10884.467444	21.379179	0.289497	42766.0
min	39.000000	39.000000	0.000000	2.000000	2.0
25%	325.000000	800.000000	30.000000	4.000000	1191.5
50%	799.000000	1670.000000	46.000000	4.100000	5179.0
75%	1999.000000	4321.250000	61.000000	4.300000	17342.2
max	77990.000000	139900.000000	100.000000	5.000000	426973.0

In [526…

```
# Create a correlation matrix of numeric columns
numeric_df = df.select_dtypes(include=['float64', 'int64'])
correlation_matrix = numeric_df.corr()

# Create a heatmap using seaborn
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f', line
plt.title('Correlation Heatmap of Numeric Features')
plt.tight_layout()
plt.show()
```



In [527…

```
from sklearn.model_selection import train_test_split
from xgboost import XGBRegressor


feature_columns = ['discount_percentage', 'actual_price']
target_column = 'rating_score'

X = df[feature_columns]
y = df[target_column]

# Split the data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)


# Initialize models
model = XGBRegressor(objective='reg:squarederror',
                     n_estimators=100, seed=123)


model.fit(X_train, y_train)
```

Out[527…

```
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, device=None, early_stopping_
rounds=None,
             enable_categorical=False, eval_metric=None, feature
_types=None,
             feature_weights=None, gamma=None, grow_policy=None,
             importance_type=None, interaction_constraints=None,
             learning_rate=None, max_bin=None, max_cat_threshold
```

In [528…

```python
def predict_rating_for_discount(discount_pct, base_actual_price=None):
    if base_actual_price is None:
        base_actual_price = df['actual_price'].median()


    calculated_discounted_price = base_actual_price * (1 - discount_pct / 10


    features = np.array([[discount_pct, calculated_discounted_price]])

    prediction = model.predict(features)[0]

    return prediction


print(f"Prediction for 10% discount: {predict_rating_for_discount(10):.4f}")
print(f"Prediction for 50% discount: {predict_rating_for_discount(50):.4f}")
```

```
Prediction for 10% discount: 4.3086
Prediction for 50% discount: 4.2334
```

In [529…

```python
discount_range = np.arange(0, 94, 1)
predicted_scores = [predict_rating_for_discount(d) for d in discount_range]
```

In [530…

```python
def calculate_rating_score_threshold(scores, method='percentile', value=90):
    """
    Calculates a threshold for rating scores based on different methods.
    """
    scores_array = np.array(scores)

    if method == 'percentile':
        if not (0 <= value <= 100):
            raise ValueError("Percentile value must be between 0 and 100.")
        return np.percentile(scores_array, value)

    elif method == 'peak_percentage':
        if not (0 <= value <= 1):
            raise ValueError(
                "Value for 'peak_percentage' must be between 0 and 1 (e.g.,
        return np.max(scores_array) * value
```

```
    elif method == 'mean_plus_std':
        return np.mean(scores_array) + (value * np.std(scores_array))
```

In [531…

```
# --- Smoothing the Predicted Scores ---
def moving_average(data, window_size):
    return np.convolve(data, np.ones(window_size)/window_size, mode='valid')


window_size = 3  # Increased window size for more aggressive smoothing
smoothed_scores = moving_average(np.array(predicted_scores), window_size)

# Adjust discount_range for smoothed scores
# The smoothed array is shorter by (window_size - 1)
smoothed_discount_range = discount_range[window_size - 1:]

# --- Find Optimal in Smoothed Range ---
optimal_smoothed_idx = np.argmax(smoothed_scores)
optimal_smoothed_discount = smoothed_discount_range[optimal_smoothed_idx]
optimal_smoothed_score = smoothed_scores[optimal_smoothed_idx]

print(f"\n--- Smoothed Optimization Results (Window Size: {window_size}) ---
print(f"Optimal Smoothed Discount Percentage: {optimal_smoothed_discount}%")
print(
    f"Predicted Smoothed Rating Score at Optimal Discount: {optimal_smoothed
```

```
--- Smoothed Optimization Results (Window Size: 3) ---
Optimal Smoothed Discount Percentage: 67%
Predicted Smoothed Rating Score at Optimal Discount: 4.4912
```

In [532…

```
rating_score_threshold_percentile = calculate_rating_score_threshold(smoothe

rating_score_threshold = rating_score_threshold_percentile

# Define the minimum width of the range
min_range_width = 5 # 5 percentage points
```

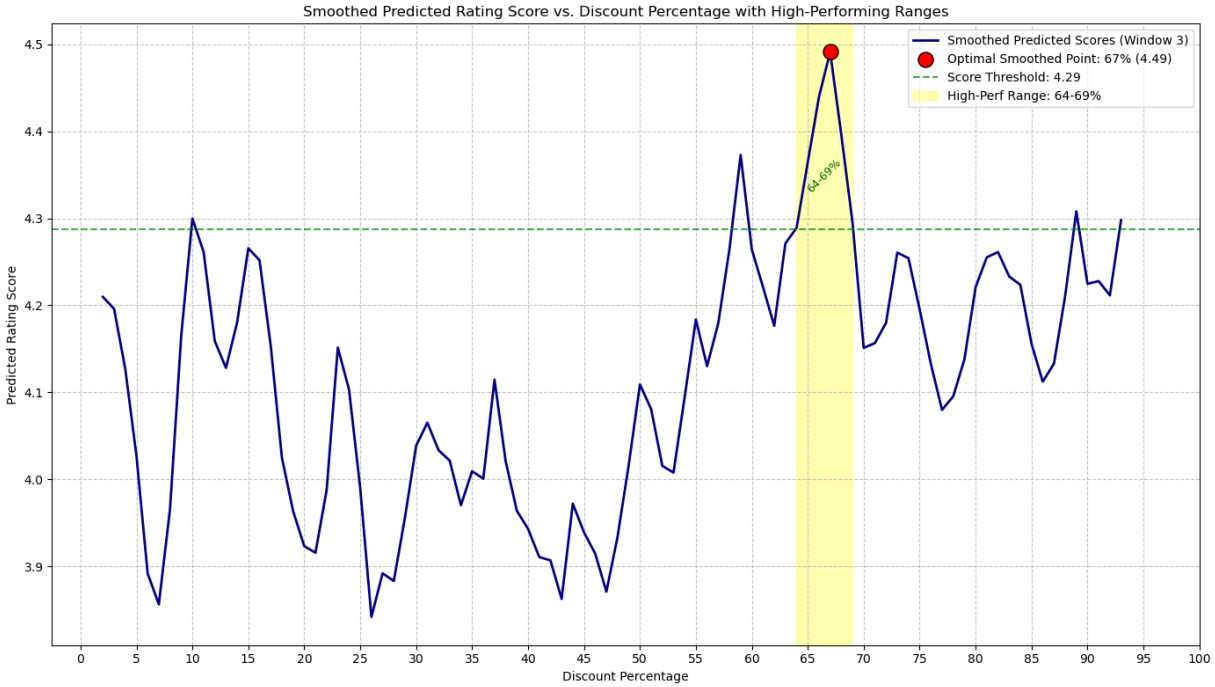
In [533…

```
high_performing_ranges = []
current_range_start_idx = -1

for i in range(len(smoothed_scores)):
    if smoothed_scores[i] >= rating_score_threshold:
        if current_range_start_idx == -1:  # Start of a new potential range
            current_range_start_idx = i
    else:  # Score dropped below threshold
        if current_range_start_idx != -1:  # End of a potential range
            current_range_end_idx = i - 1
            range_start_pct = smoothed_discount_range[current_range_start_id
            range_end_pct = smoothed_discount_range[current_range_end_idx]

            if (range_end_pct - range_start_pct + 1) >= min_range_width:
                # Calculate average score for this specific range
                avg_score_in_range = np.mean(
                    smoothed_scores[current_range_start_idx: current_range_e
                high_performing_ranges.append({
                    'start_pct': range_start_pct,
```

```
                    'end_pct': range_end_pct,
                    'avg_score': avg_score_in_range
                })
            current_range_start_idx = -1  # Reset

# Handle the last range if it extends to the end of the discount_range
if current_range_start_idx != -1:
    current_range_end_idx = len(smoothed_scores) - 1
    range_start_pct = smoothed_discount_range[current_range_start_idx]
    range_end_pct = smoothed_discount_range[current_range_end_idx]

    if (range_end_pct - range_start_pct + 1) >= min_range_width:
        avg_score_in_range = np.mean(
            smoothed_scores[current_range_start_idx: current_range_end_idx +
        high_performing_ranges.append({
            'start_pct': range_start_pct,
            'end_pct': range_end_pct,
            'avg_score': avg_score_in_range
        })

print(
    f"\n--- High-Performing Discount Ranges (Smoothed Score >= {rating_score
if high_performing_ranges:
    # Sort by average score for clarity
    high_performing_ranges.sort(key=lambda x: x['avg_score'], reverse=True)
    for r in high_performing_ranges:
        print(
            f"Range: {r['start_pct']}% - {r['end_pct']}% | Average Predicted
else:
    print("No high-performing ranges found with the given criteria.")


# --- Visualization of the smoothed curve and identified ranges ---
plt.figure(figsize=(14, 8))
plt.plot(smoothed_discount_range, smoothed_scores, color='darkblue',
         linewidth=2, label=f'Smoothed Predicted Scores (Window {window_size
plt.scatter(optimal_smoothed_discount, optimal_smoothed_score, color='red',
            label=f'Optimal Smoothed Point: {optimal_smoothed_discount}% ({o

# Plot the threshold line
plt.axhline(y=rating_score_threshold, color='green', linestyle='--',
            alpha=0.7, label=f'Score Threshold: {rating_score_threshold:.2f}

# Highlight the identified ranges
for r in high_performing_ranges:
    start_idx = np.where(smoothed_discount_range == r['start_pct'])[0][0]
    end_idx = np.where(smoothed_discount_range == r['end_pct'])[0][0]
    plt.axvspan(r['start_pct'], r['end_pct'], color='yellow', alpha=0.3,
                label=f'High-Perf Range: {r["start_pct"]}-{r["end_pct"]}%' i
    plt.text(r['start_pct'] + (r['end_pct'] - r['start_pct']) / 2, rating_sc
             f'{r["start_pct"]}-{r["end_pct"]}%', horizontalalignment='cente


plt.title('Smoothed Predicted Rating Score vs. Discount Percentage with High
plt.xlabel('Discount Percentage')
plt.ylabel('Predicted Rating Score')
```

```
plt.grid(True, linestyle='--', alpha=0.7)
plt.xticks(np.arange(0, 101, 5))
plt.legend()
plt.tight_layout()
plt.show()
```

```
--- High-Performing Discount Ranges (Smoothed Score >= 4.2872, Min Width: 5%) ---
Range: 64% - 69% | Average Predicted Score: 4.3792
```



A/B Testing

Hypotheses for Independent Samples t-test

Null Hypothesis (H_0)

There is no statistically significant difference in the average **rating_score** between products offered with a **63% - 68% discount** (**Treatment A group**) and products offered with other discount percentages (**Control group**).

Mathematically:

$\mu_{Treatment\_A} = \mu_{Control}$

Alternative Hypothesis (H_1)

There is a statistically significant difference in the average **rating_score** between products offered with a **63% - 68% discount** (**Treatment A group**) and products offered with other discount percentages (**Control group**).

Mathematically:

$\mu_{Treatment\_A} \neq \mu_{Control}$

Purpose of the Test

The purpose of the statistical test is to determine if there is sufficient evidence to reject the **null hypothesis** in favor of the **alternative hypothesis**.

In [534…
```
from statsmodels.formula.api import ols
from statsmodels.stats.anova import anova_lm
from statsmodels.stats.multicomp import pairwise_tukeyhsd
from scipy import stats
```

In [535…
```
df = pd.read_csv('amazon_updated.csv')

# Cleaning Data

df['discount_percentage'] = df['discount_percentage'].str.replace(
    "%", "").astype(int)

df['actual_price'] = df['actual_price'].str.replace(",", "").astype(float)

df['rating_count'] = df['rating_count'].astype(str).str.replace(
    ',', '').pipe(pd.to_numeric, errors='coerce')

df['rating'] = pd.to_numeric(df['rating'], errors='coerce')


df['rating'] = df['rating'].fillna(df['rating'].median())
df['rating_count'] = df['rating_count'].fillna(df['rating_count'].median())

mean_rating = df['rating'].mean()
min_ratings = 15

df['rating_score'] = (df['rating_count'] * df['rating'] +
                      min_ratings * mean_rating) / (df['rating_count'] + mir
```

In [536…
```
def assign_group(discount_pct):
    """
    Assigns a group to a product. 'Treatment_A' if discount_pct is between 6
    otherwise 'Control'.
    """
    if 64 <= discount_pct <= 68:
        return 'Treatment_A'
    else:
        return 'Control'


# Apply the function to create the 'group' column
df['group'] = df['discount_percentage'].apply(assign_group)

print("\n--- Descriptive Statistics by Group ---")
print(df.groupby('group')['rating_score'].agg(['mean', 'std', 'count']))
```

```
--- Descriptive Statistics by Group ---
                mean       std  count
group
Control      4.094876  0.270333   1381
Treatment_A  4.212945  0.142661     83
```

In [537…

```
# --- Perform Statistical Analysis ---
alpha = 0.05  # Standard significance level

# Check Homogeneity of Variances (Levene's Test)
print("\n--- Checking Homogeneity of Variances (Levene's Test) ---")

unique_groups = df['group'].unique()
groups_data = [df['rating_score'][df['group'] == g] for g in unique_groups]
groups_data = [g_data for g_data in groups_data if len(
    g_data) > 0]  # Filter out empty groups

# Ensure there are exactly two groups with data for a two-sample t-test
if len(groups_data) == 2:
    stat, p_levene = stats.levene(*groups_data)
    print(f'Levene Test: Statistics={stat:.3f}, p={p_levene:.3f}')
    if p_levene > alpha:
        print('  Variances are equal (fail to reject H0). A standard indeper
        equal_var_assumption = True
    else:
        print('  Variances are not equal (reject H0). Welch\'s t-test will b
        equal_var_assumption = False

    # Perform Independent Samples t-test (either standard or Welch's)
    print("\n--- Performing Independent Samples t-test ---")
    group1_data = groups_data[0]
    group2_data = groups_data[1]

    # The ttest_ind function automatically performs Welch's t-test if equal_
    t_stat, p_ttest = stats.ttest_ind(
        group1_data, group2_data, equal_var=equal_var_assumption)
    print(f't-test: t-statistic={t_stat:.3f}, p={p_ttest:.3f}')

    if p_ttest < alpha:
        print(
            f'  Reject the null hypothesis: There is a significant differenc
    else:
        print(
            f'  Fail to reject the null hypothesis: No significant differenc

    # Optionally, print the group names for clarity
    print(f"\nComparing '{unique_groups[0]}' vs '{unique_groups[1]}'")
    print(f"Mean '{unique_groups[0]}': {group1_data.mean():.3f}")
    print(f"Mean '{unique_groups[1]}': {group2_data.mean():.3f}")

elif len(groups_data) < 2:
    print("Not enough groups with data to perform statistical tests effectiv
else:
    print("More than two groups detected for a two-group comparison. This co
```

```
--- Checking Homogeneity of Variances (Levene's Test) ---
Levene Test: Statistics=22.291, p=0.000
  Variances are not equal (reject H0). Welch's t-test will be performed, whi
ch does not assume equal variances.

--- Performing Independent Samples t-test ---
t-test: t-statistic=6.838, p=0.000
  Reject the null hypothesis: There is a significant difference between the
means of the two groups (p < 0.05).

Comparing 'Treatment_A' vs 'Control'
Mean 'Treatment_A': 4.213
Mean 'Control': 4.095
```