THE VIDEO GAME DATASET

The video game dataset provides useful insights into the global video game sales. It contains up-to-date statistics on the global sales performance and popularity of numerous video games. Name, platform, year of release, genre, publisher, and sales in North America, Europe, Japan, and other territories are all included in the data. It also includes critic and user scores and ratings, such as the average critic score, number of critics reviewed, average user score, number of people who reviewed, developer, and rating.

OBJECTIVE

- The aim of this task is to predict the worldwide sales performance of many video games.

Import the required libraries

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

In [2]:

```
#Read the dataset and assign to a variable
video_games_model = pd.read_csv('Video_Games.csv')

#print the first five rows
video_games_model.head()
```

Out[2]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sales	JP_Sal
0	Wii Sports	Wii	2006.0	Sports	Nintendo	41.36	28.96	3.
1	Super Mario Bros.	NES	1985.0	Platform	Nintendo	29.08	3.58	6.
2	Mario Kart Wii	Wii	2008.0	Racing	Nintendo	15.68	12.76	3.
3	Wii Sports Resort	Wii	2009.0	Sports	Nintendo	15.61	10.93	3.
4	Pokemon Red/Pokemon Blue	GB	1996.0	Role-Playing	Nintendo	11.27	8.89	10.

EXPLORATORY DATA ANALYSIS

In [3]:

```
# check the summary information of the data to get an understanding of features, dat
video_games_model.info()
```

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

- The summary information of the data shows the total number of columns and rows in the dataset, as well as the presence of missing values in some columns, the inconsistencies and datatypes

In [4]:

```
# Get the sum of missing values in the columns. This can also be derived with 'video
video_games_model.isna().sum()
```

Out[4]:

```
Name                  2
Platform              0
Year_of_Release     269
Genre                 2
Publisher            54
NA_Sales              0
EU_Sales              0
JP_Sales              0
Other_Sales           0
Global_Sales          0
Critic_Score       8582
Critic_Count       8582
User_Score         6704
User_Count         9129
Developer          6623
Rating             6769
dtype: int64
```

- The number of missing values above shows the amount of cleaning that will be done on the data before training the model

In [5]:

```
# Checking for duplicate records
video_games_model.duplicated().sum()
```

Out[5]:

```
0
```

- The dataset have no duplicate entries.

In [6]:

```
# descriptive statistics of the data
'''
As '.describe()' by default works on only numeric variables,
I used the ' include='all' to include the categorical variables and .T to transpose
the output for better view
 '''
video_games_model.describe( include='all').T
```

Out[6]:

	count	unique	top	freq	mean	std	min	25%
Name	16717	11562	Need for Speed: Most Wanted	12	NaN	NaN	NaN	NaN
Platform	16719	31	PS2	2161	NaN	NaN	NaN	NaN
Year_of_Release	16450.0	NaN	NaN	NaN	2006.487356	5.878995	1980.0	2003.0
Genre	16717	12	Action	3370	NaN	NaN	NaN	NaN
Publisher	16665	581	Electronic Arts	1356	NaN	NaN	NaN	NaN
NA_Sales	16719.0	NaN	NaN	NaN	0.26333	0.813514	0.0	0.0
EU_Sales	16719.0	NaN	NaN	NaN	0.145025	0.503283	0.0	0.0
JP_Sales	16719.0	NaN	NaN	NaN	0.077602	0.308818	0.0	0.0
Other_Sales	16719.0	NaN	NaN	NaN	0.047332	0.18671	0.0	0.0
Global_Sales	16719.0	NaN	NaN	NaN	0.533543	1.547935	0.01	0.06
Critic_Score	8137.0	NaN	NaN	NaN	68.967679	13.938165	13.0	60.0
Critic_Count	8137.0	NaN	NaN	NaN	26.360821	18.980495	3.0	12.0
User_Score	10015	96	tbd	2425	NaN	NaN	NaN	NaN
User_Count	7590.0	NaN	NaN	NaN	162.229908	561.282326	4.0	10.0
Developer	10096	1696	Ubisoft	204	NaN	NaN	NaN	NaN
Rating	9950	8	E	3991	NaN	NaN	NaN	NaN

- A brief summary of the descriptive statistics:
  - The year of release ranges from 1980 -2020 (4 deacdes).
  - Need for Speed: Most Wanted was the most sold video game name within the 4 decades, PS2 was the most sold platform, Action was the most occuring Genre in the dataset, the top Publisher was Electronic Arts, top Developer was Ubisoft and top Rating was E.
  - Among the independent variables, North America had the overall maximum sales of 41.360000 while they all had a minimum sales of 0.000000.
  - The Critic score ranges from 13 - 98, Critic count 3 - 113, and User score 4 - 10665.

In [7]:

```
#checking the value counts for all the categorical variables
object_data = video_games_model.select_dtypes('object')
for col in object_data:
    print(f'{col}\n')
    print(f'{object_data[col].value_counts()} \n')
```

```
Name

Need for Speed: Most Wanted                          12
FIFA 14                                               9
Ratatouille                                           9
LEGO Marvel Super Heroes                              9
Madden NFL 07                                         9
                                                     ..
Jewels of the Tropical Lost Island                    1
Sherlock Holmes and the Mystery of Osborne House      1
The King of Fighters '95 (CD)                         1
Megamind: Mega Team Unite                             1
Haitaka no Psychedelica                               1
Name: Name, Length: 11562, dtype: int64

Platform

PS2     2161
DS      2152
PS3     1331
Wii     1320
X360    1262
PSP     1209
PS      1197
PC       974
XB       824
GBA      822
GC       556
3DS      520
PSV      432
PS4      393
N64      319
XOne     247
SNES     239
SAT      173
WiiU     147
2600     133
NES       98
GB        98
DC        52
GEN       29
NG        12
SCD        6
WS         6
3DO        3
TG16       2
GG         1
PCFX       1
Name: Platform, dtype: int64

Genre

Action          3370
Sports          2348
Misc            1750
Role-Playing    1500
Shooter         1323
Adventure       1303
Racing          1249
Platform         888
Simulation       874
```

```
Fighting         849
Strategy         683
Puzzle           580
Name: Genre, dtype: int64

Publisher

Electronic Arts                 1356
Activision                       985
Namco Bandai Games               939
Ubisoft                          933
Konami Digital Entertainment     834
                                ...
Valve                              1
ITT Family Games                   1
Elite                              1
Evolution Games                    1
Red Flagship                       1
Name: Publisher, Length: 581, dtype: int64

User_Score

tbd    2425
7.8     324
8       290
8.2     282
8.3     254
       ...
1.1       2
1.9       2
9.6       2
0         1
9.7       1
Name: User_Score, Length: 96, dtype: int64

Developer

Ubisoft                                     204
EA Sports                                   172
EA Canada                                   167
Konami                                      162
Capcom                                      139
                                           ...
Genki, Kojima Productions                     1
Warner Bros. Interactive Entertainment        1
THQ, Altron                                   1
Netherock Ltd.                                1
Interchannel-Holon                            1
Name: Developer, Length: 1696, dtype: int64

Rating

E       3991
T       2961
M       1563
E10+    1420
EC         8
K-A        3
RP         3
AO         1
```

```
Name: Rating, dtype: int64
```

- The value counts for all the categorical variables.

DATA CLEANING

Name Column

In [8]:

```
#subsetting the data to get the rows of the missing value in 'Name Column'
video_games_model[video_games_model['Name'].isnull()]
```

Out[8]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sales	JP_Sale
659	NaN	GEN	1993.0	NaN	Acclaim Entertainment	1.78	0.53	0.0
14246	NaN	GEN	1993.0	NaN	Acclaim Entertainment	0.00	0.00	0.0

In [9]:

```
# Ascertaining the mode of 'Name Column'
video_games_model['Name'].mode() # I am using mode because the Name column is a cate
```

Out[9]:

```
0    Need for Speed: Most Wanted
Name: Name, dtype: object
```

In [10]:

```
# Replacing the two null values in 'Name Column' with the mode of the column, and se
video_games_model['Name'].fillna('Need for Speed: Most Wanted', inplace= True)
```

In [11]:

```
# checking for empty strings because missing values does not equal blank spaces
video_games_model[video_games_model['Name'] == ' ']
```

Out[11]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sales	JP_Sales	Other_

Year of Release Column

In [12]:

```
1 video_games_model['Year_of_Release'] # a quick glance at the column
```

Out[12]:

```
0        2006.0
1        1985.0
2        2008.0
3        2009.0
4        1996.0
          ...
16714    2016.0
16715    2006.0
16716    2016.0
16717    2003.0
16718    2016.0
Name: Year_of_Release, Length: 16719, dtype: float64
```

In [13]:

```
1 video_games_model[video_games_model['Year_of_Release'].isnull()]
```

Out[13]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sale
183	Madden NFL 2004	PS2	NaN	Sports	Electronic Arts	4.26	0.2
377	FIFA Soccer 2004	PS2	NaN	Sports	Electronic Arts	0.59	2.3
456	LEGO Batman: The Videogame	Wii	NaN	Action	Warner Bros. Interactive Entertainment	1.80	0.9
475	wwe Smackdown vs. Raw 2006	PS2	NaN	Fighting	NaN	1.57	1.0
609	Space Invaders	2600	NaN	Shooter	Atari	2.36	0.1
...	...	...	...	...	...	...	.
16376	PDC World Championship Darts 2008	PSP	NaN	Sports	Oxygen Interactive	0.01	0.0
16409	Freaky Flyers	GC	NaN	Racing	Unknown	0.01	0.0
16452	Inversion	PC	NaN	Shooter	Namco Bandai Games	0.01	0.0
16462	Hakuouki: Shinsengumi Kitan	PS3	NaN	Adventure	Unknown	0.01	0.0
16526	Virtua Quest	GC	NaN	Role-Playing	Unknown	0.01	0.0

269 rows × 16 columns

In [14]:

```
#subsetting the data to get the rows of the missing value in 'Year Column'
video_games_model[video_games_model['Year_of_Release'].isnull()]
```

Out[14]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sales	JP_Sales
183	Madden NFL 2004	PS2	NaN	Sports	Electronic Arts	4.26	0.26	0.01
377	FIFA Soccer 2004	PS2	NaN	Sports	Electronic Arts	0.59	2.36	0.04
456	LEGO Batman: The Videogame	Wii	NaN	Action	Warner Bros. Interactive Entertainment	1.80	0.97	0.00
475	wwe Smackdown vs. Raw 2006	PS2	NaN	Fighting	NaN	1.57	1.02	0.00
609	Space Invaders	2600	NaN	Shooter	Atari	2.36	0.14	0.00

In [15]:

```
# Replacing the null values in 'Year Column' with the median of the column, and sett
# I chose median because it is not affected by outliers unlike mean
video_games_model['Year_of_Release'].fillna(video_games_model['Year_of_Release'].med
```

In [16]:

```
# Casting the Year_of_Release datatype to Integer
video_games_model['Year_of_Release'] = video_games_model['Year_of_Release'].astype(i
```

In [17]:

```
# checking for empty strings because missing values does not equal blank spaces
video_games_model[video_games_model['Year_of_Release'] == ' ']
```

Out[17]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sales	JP_Sales	Other

Genre Column

In [18]:

```
#subsetting the data to get the rows of the missing value in 'Genre Column'
video_games_model[video_games_model['Genre'].isnull()]
```

Out[18]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sales	JP_Sa
659	Need for Speed: Most Wanted	GEN	1993	NaN	Acclaim Entertainment	1.78	0.53	0
14246	Need for Speed: Most Wanted	GEN	1993	NaN	Acclaim Entertainment	0.00	0.00	0

In [19]:

```
# Ascertaining the mode of 'Name Column', I am using mode because the Genre column i
video_games_model['Genre'].mode()
```

Out[19]:

```
0    Action
Name: Genre, dtype: object
```

In [20]:

```
# Replacing the null values in 'Genre Column' with the mode of the column, and setti

video_games_model['Genre'].fillna('Action', inplace= True)
```

In [21]:

```
# checking for empty strings because missing values does not equal blank spaces
video_games_model[video_games_model['Genre'] == ' ']
```

Out[21]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sales	JP_Sales	Other

Publisher Column

In [22]:

```
#subsetting the data to get the rows of the missing value in 'Publisher Column'
video_games_model[video_games_model['Publisher'].isnull()]
```

Out[22]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sales	JP_Sales	Otl
475	wwe Smackdown vs. Raw 2006	PS2	2007	Fighting	NaN	1.57	1.02	0.00	
1301	Triple Play 99	PS	2007	Sports	NaN	0.81	0.55	0.00	
1667	Shrek / Shrek 2 2-in-1 Gameboy Advance Video	GBA	2007	Misc	NaN	0.87	0.32	0.00	
2212	Bentley's Hackpack	GBA	2005	Misc	NaN	0.67	0.25	0.00	
3148	Nicktoons Collection: Game Boy	GBA	2004	Misc	NaN	0.46	0.17	0.00	

In [23]:

```
# Ascertaining the mode of 'Publisher Column', I am using mode because Publisher  co
video_games_model['Publisher'].mode()
```

Out[23]:

```
0    Electronic Arts
Name: Publisher, dtype: object
```

In [24]:

```
# Replacing the null values in 'Publisher Column' with the mode of the column, and s

video_games_model['Publisher'].fillna('Electronic Arts', inplace= True)
```

In [25]:

```
# checking for empty strings because missing values does not equal blank spaces
video_games_model[video_games_model['Publisher'] == ' ']
```

Out[25]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sales	JP_Sales	Other_

Critic Score Column

In [26]:

```
"""
Replacing the null values in 'Critic_Score Column' with 0, and setting inplace =True
I chose 0 because there is no 0 value coupled with the fact that Critic_Score and Cr
of missing values. As the inputs were not provided in both culumns, they should be z


I tried replacing the missing values with the median and it increased the outliers.
"""
video_games_model['Critic_Score'].fillna(0, inplace= True)
```

Critic Count Column

In [27]:

```
'''
Replacing the null values in 'Critic_Count Column' with 0, and setting inplace =True
there is no 0 value coupled with the fact that Critic_Count and Critic_Score have sa
were not provided in both culumns, they should be zero.
'''
video_games_model['Critic_Count'].fillna(0, inplace= True)
```

User Score Column

In [28]:

```
# Replacing the object data type in the entries to missing values
video_games_model['User_Score'].replace('tbd', np.nan, inplace= True)
```

In [29]:

```
# Casting User_Score column from object data type to float
video_games_model['User_Score'] = video_games_model['User_Score'].astype(float)
```

In [30]:

```
# Replacing the null values in 'User_Score Column' with the median of the column, an
# permanent. I chose median because it is not affected by outliers and there is 0 in
video_games_model['User_Score'].fillna(video_games_model['User_Score'].median(), inp
```

User Count Column

In [31]:

```
# Checking the statistics of the data
video_games_model['User_Count'].describe()
```

Out[31]:

```
count     7590.000000
mean       162.229908
std        561.282326
min          4.000000
25%         10.000000
50%         24.000000
75%         81.000000
max      10665.000000
Name: User_Count, dtype: float64
```

In [32]:

```
# Replacing the null values in 'User_Count Column' with 0, and setting inplace =True
# permanent. I chose 0 because there is no 0 in the column and believe that wherever
video_games_model['User_Count'].fillna(0, inplace= True)
```

Developer Column

In [33]:

```
# Ascertaining the mode of 'Developer Column', I am using mode because Developer col
video_games_model['Developer'].mode()
```

Out[33]:

```
0    Ubisoft
Name: Developer, dtype: object
```

In [34]:

```
# Replacing the null values in 'Developer Column' with the mode of the column, and s

video_games_model['Developer'].fillna('Ubisoft', inplace= True)
```

Rating Column

In [35]:

```
# Filling the missing values with unknown
video_games_model['Rating'].fillna('Unknown', inplace= True)
```

Verifying the data cleaning

In [36]:

```
1 video_games_model.isnull().sum()
```

Out[36]:

```
Name              0
Platform          0
Year_of_Release   0
Genre             0
Publisher         0
NA_Sales          0
EU_Sales          0
JP_Sales          0
Other_Sales       0
Global_Sales      0
Critic_Score      0
Critic_Count      0
User_Score        0
User_Count        0
Developer         0
Rating            0
dtype: int64
```

In [37]:

```
1 video_games_model['Platform'].value_counts()
```

Out[37]:

```
PS2     2161
DS      2152
PS3     1331
Wii     1320
X360    1262
PSP     1209
PS      1197
PC       974
XB       824
GBA      822
GC       556
3DS      520
PSV      432
PS4      393
N64      319
XOne     247
SNES     239
SAT      173
WiiU     147
2600     133
NES       98
GB        98
DC        52
GEN       29
NG        12
SCD        6
WS         6
3DO        3
TG16       2
GG         1
PCFX       1
Name: Platform, dtype: int64
```

DATA VISUALISATION

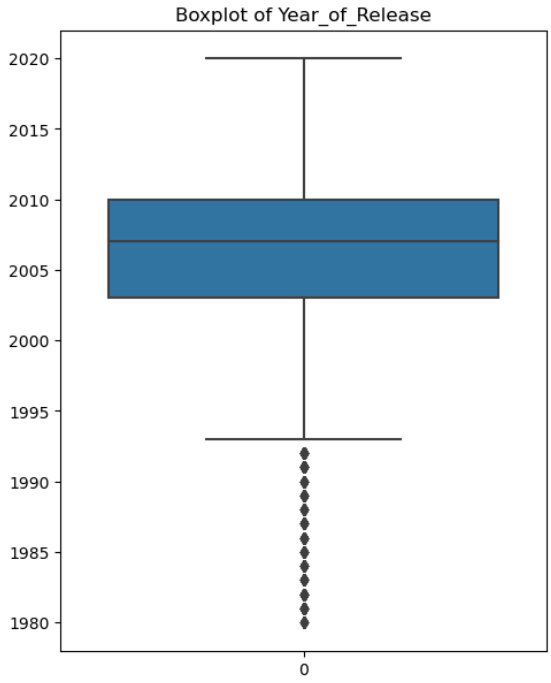
Univariate Analysis: Numeric Features

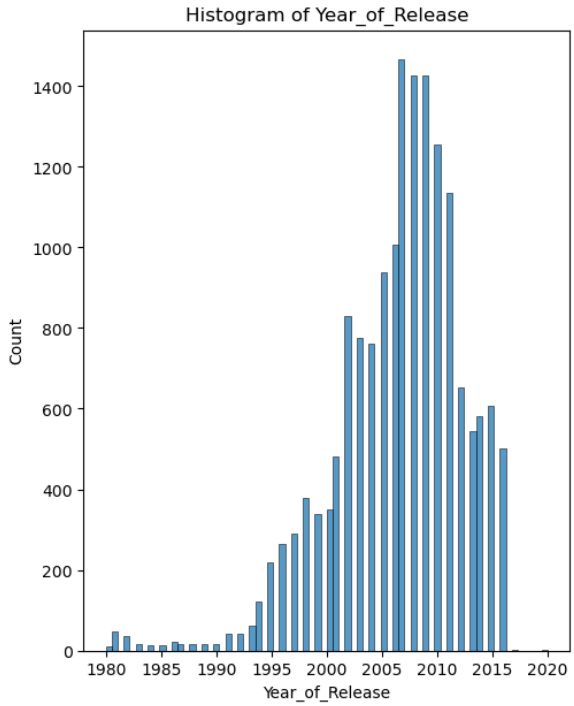
In [38]:

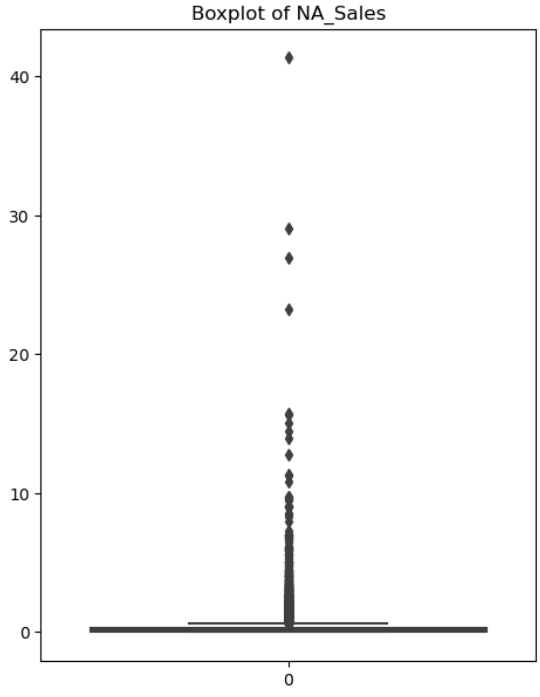
```
# A function that will create both boxplot and histogram the numeric variables
def boxplot_hist(data):
    num_data = data.select_dtypes(include='number').columns
    for feature in num_data:
        fig, axs = plt.subplots(ncols=2, figsize=(12, 7))
        sns.boxplot(data=data[feature], ax=axs[0])
        sns.histplot(data[feature], ax=axs[1])
        axs[0].set_title(f'Boxplot of {feature}')
        axs[1].set_title(f'Histogram of {feature}')
        plt.show()

```

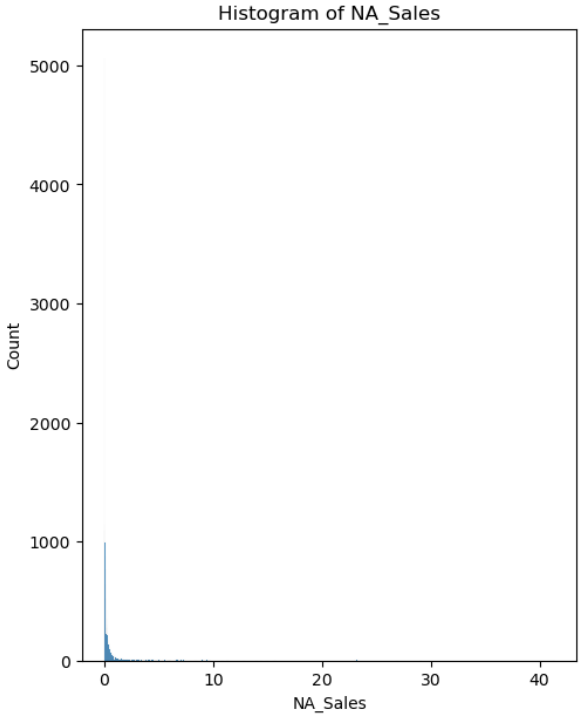
In [39]:

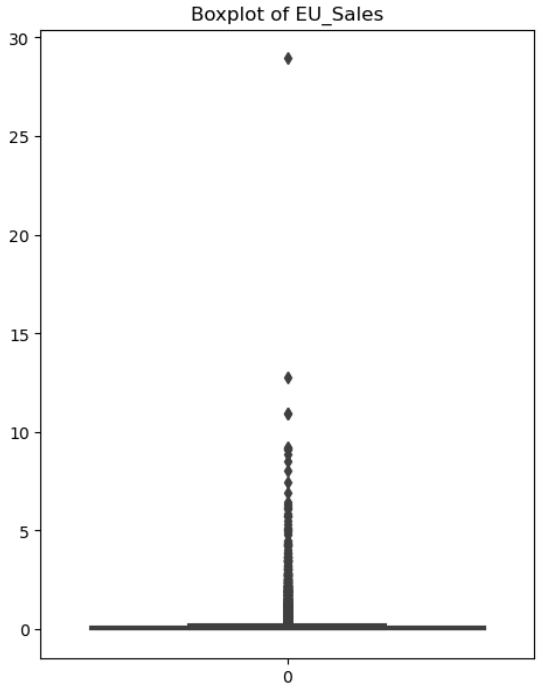
```
1 boxplot_hist(video_games_model)
```

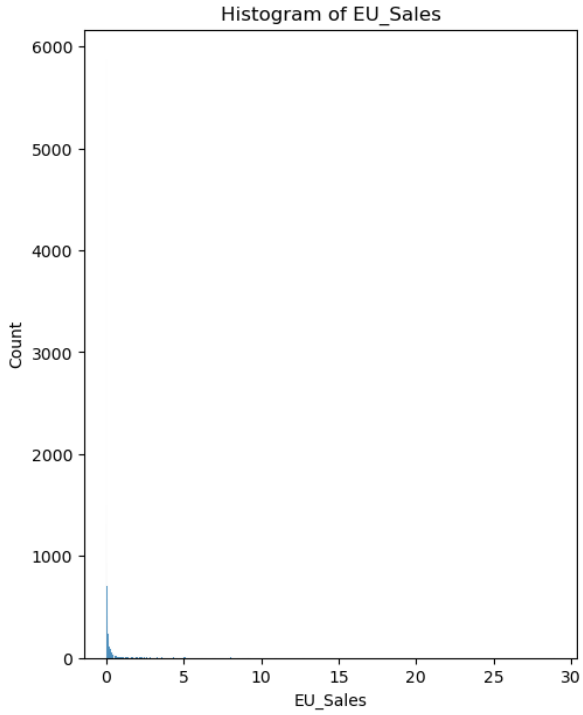


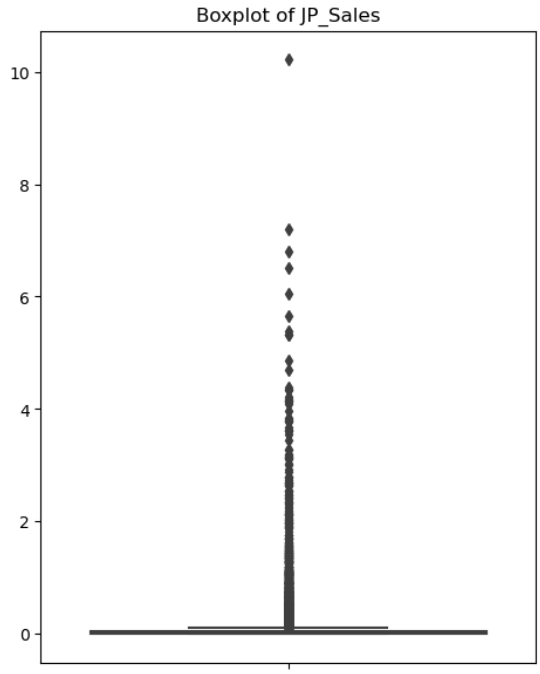


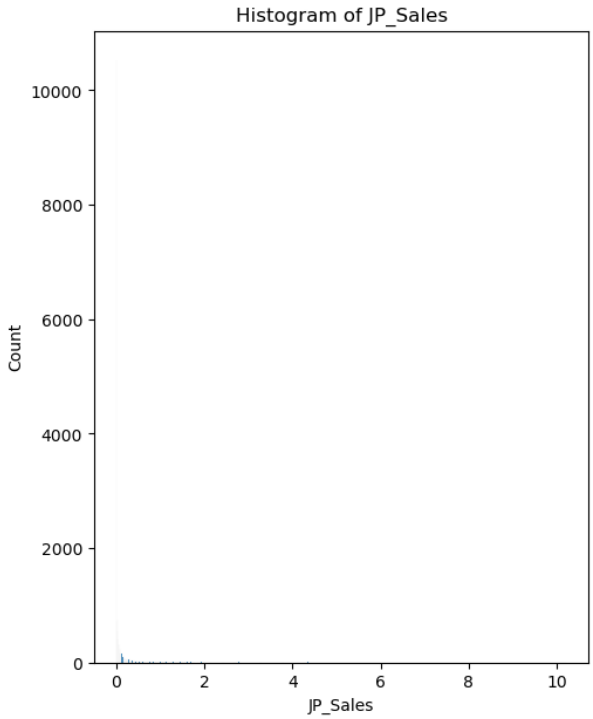


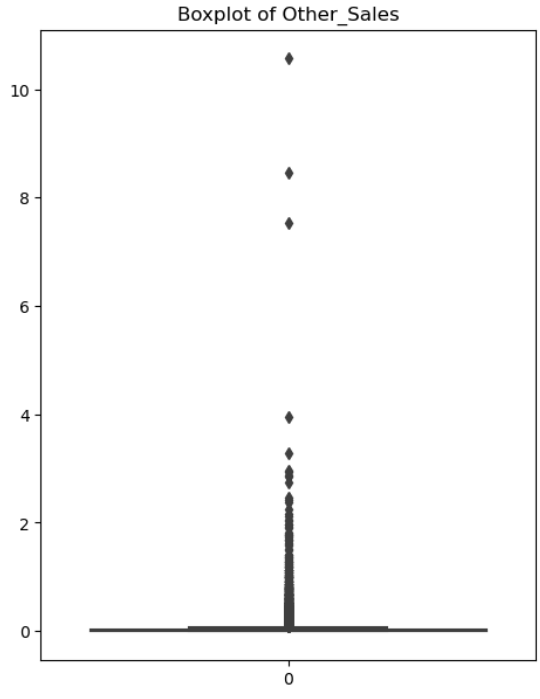


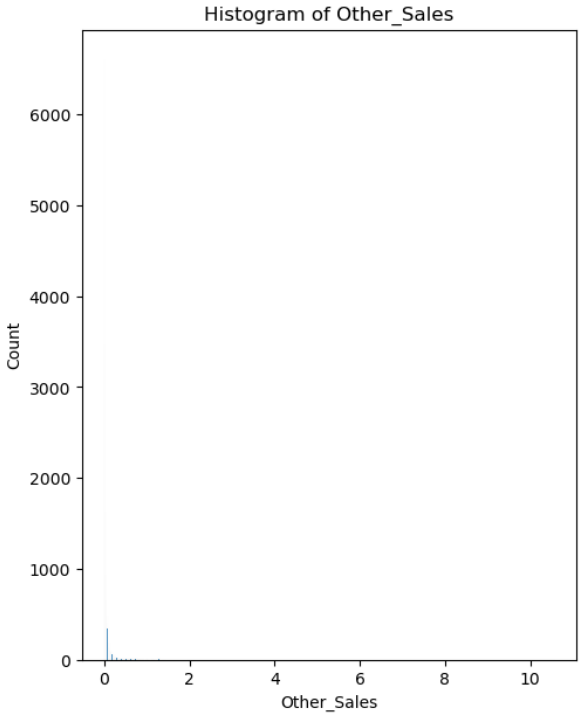

Boxplot of EU_Sales
30
25
20
15
10
5
0
0

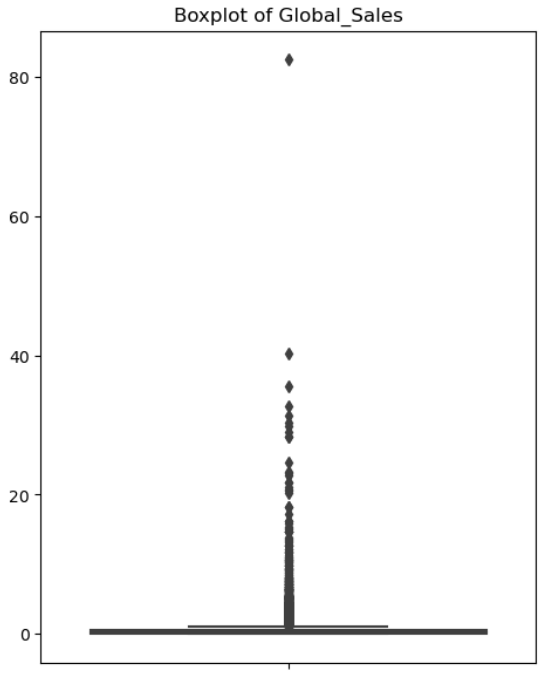

Histogram of EU_Sales
6000
5000
4000
3000
2000
1000
0
Count
0
5
10
15
20
25
30
EU_Sales

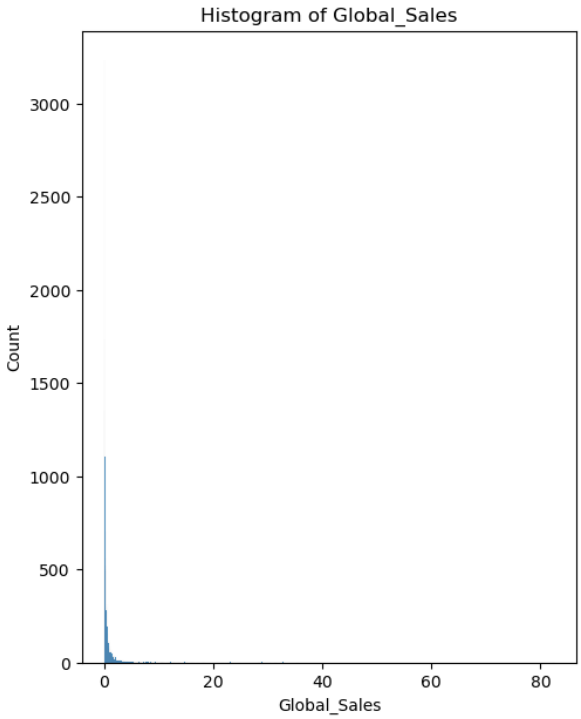

Boxplot of JP_Sales
10
8
6
4
2
0
0

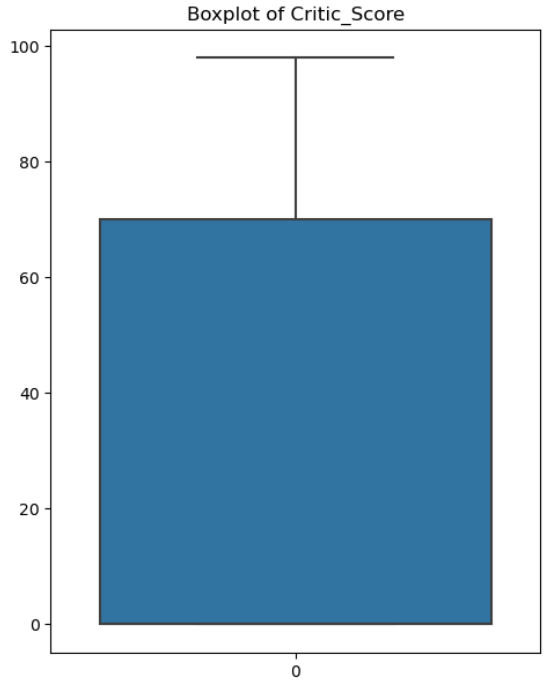

Histogram of JP_Sales
10000
8000
6000
4000
2000
0
Count
0
2
4
6
8
10
JP_Sales

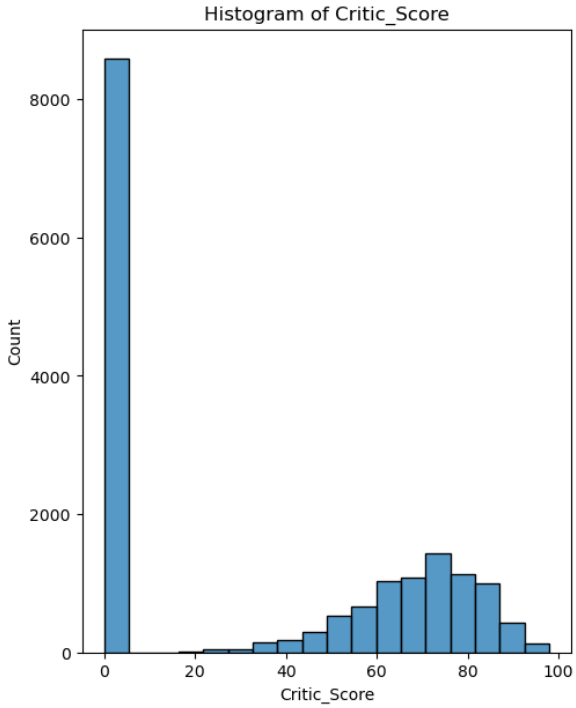

Boxplot of Other_Sales
10
8
6
4
2
0
0

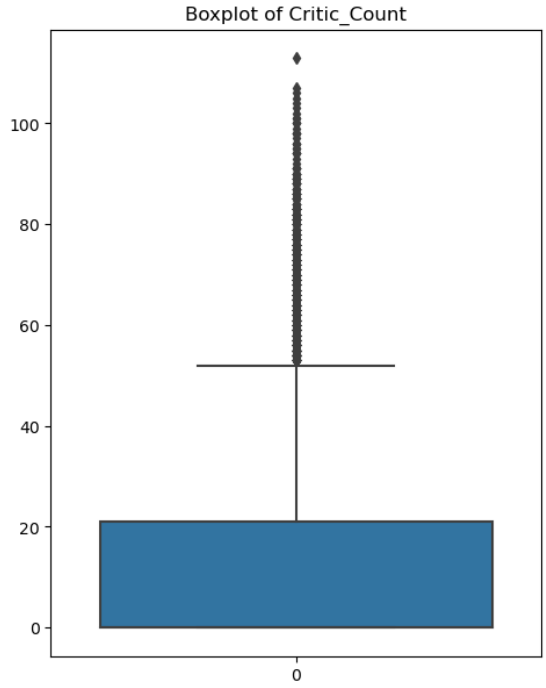

Histogram of Other_Sales
Count
6000
5000
4000
3000
2000
1000
0
0
2
4
6
8
10
Other_Sales

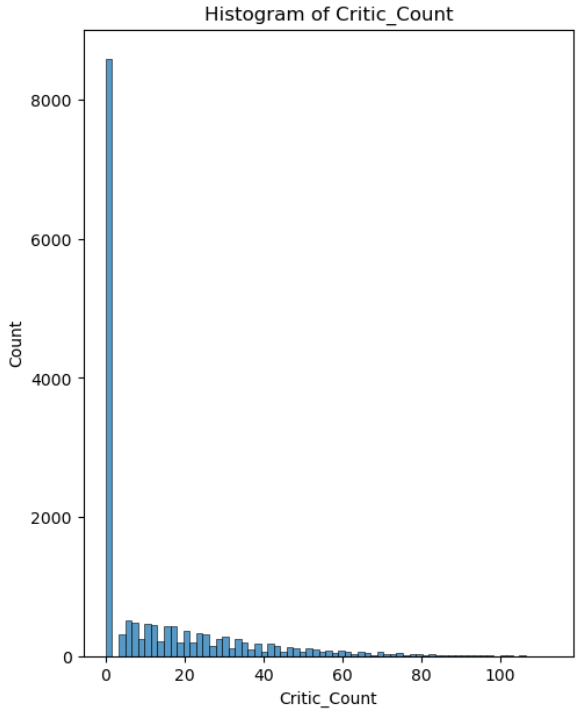

Boxplot of Global_Sales
80
60
40
20
0
0


Histogram of Global_Sales
Count
3000
2500
2000
1500
1000
500
0
0
20
40
60
80
Global_Sales










```
1 cat_data = video_games_model.select_dtypes(include='object')
```

```
cat_data
```

Out[41]:

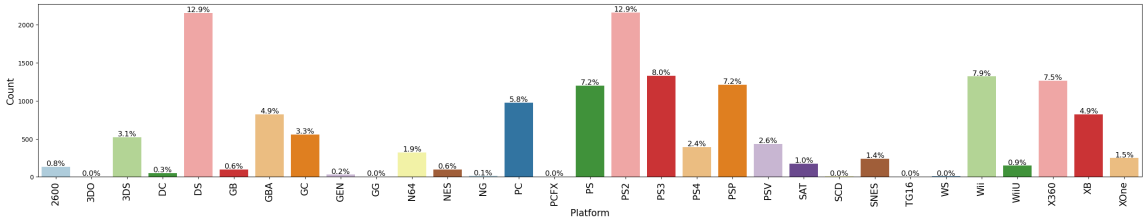
	Name	Platform	Genre	Publisher	Developer	Rating
0	Wii Sports	Wii	Sports	Nintendo	Nintendo	E
1	Super Mario Bros.	NES	Platform	Nintendo	Ubisoft	Unknown
2	Mario Kart Wii	Wii	Racing	Nintendo	Nintendo	E
3	Wii Sports Resort	Wii	Sports	Nintendo	Nintendo	E
4	Pokemon Red/Pokemon Blue	GB	Role-Playing	Nintendo	Ubisoft	Unknown
...	...	...	...	...	...	...
16714	Samurai Warriors: Sanada Maru	PS3	Action	Tecmo Koei	Ubisoft	Unknown
16715	LMA Manager 2007	X360	Sports	Codemasters	Ubisoft	Unknown
16716	Haitaka no Psychedelica	PSV	Adventure	Idea Factory	Ubisoft	Unknown
16717	Spirits & Spells	GBA	Platform	Wanadoo	Ubisoft	Unknown
16718	Winning Post 8 2016	PSV	Simulation	Tecmo Koei	Ubisoft	Unknown

16719 rows × 6 columns

```
def cat_barplot(data, feature, perc=False, n=None):
    total = len(data[feature])
    fig, ax = plt.subplots(figsize=(n+1 if n else len(data[feature].unique())+1, 5))
    sns.countplot(x=feature, data=data, order=data[feature].value_counts().index[:n]
    ax.set_xticklabels(ax.get_xticklabels(), rotation=90, fontsize=15)
    ax.set_xlabel(feature, fontsize=15)
    ax.set_ylabel('Count', fontsize=15)
    for p in ax.patches:
        label = f"{p.get_height() / total * 100:.1f}%" if perc else int(p.get_height
        ax.annotate(label, (p.get_x() + p.get_width() / 2, p.get_height()), ha='cent
    plt.show()

```

In [43]:

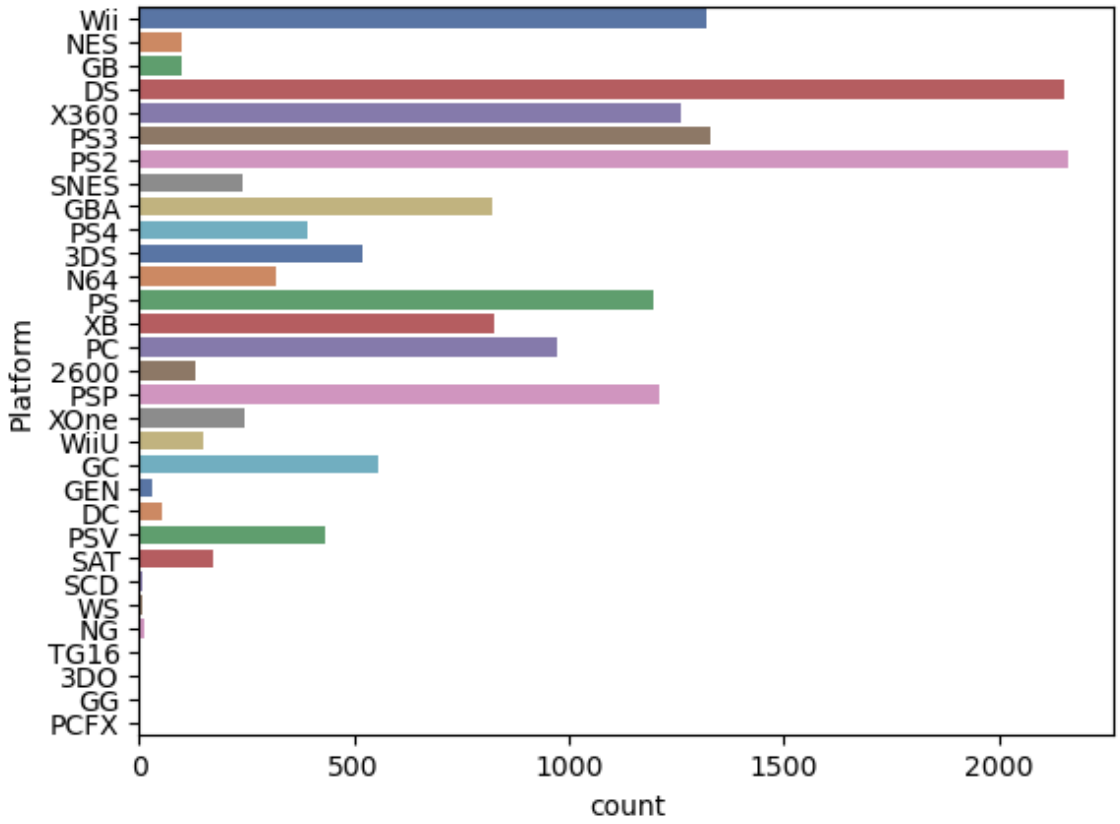
```
cat_barplot(data=cat_data, feature='Platform', perc= True, n= None )
```



- **DS and PS2 are the most used platforms followed by PS3.**

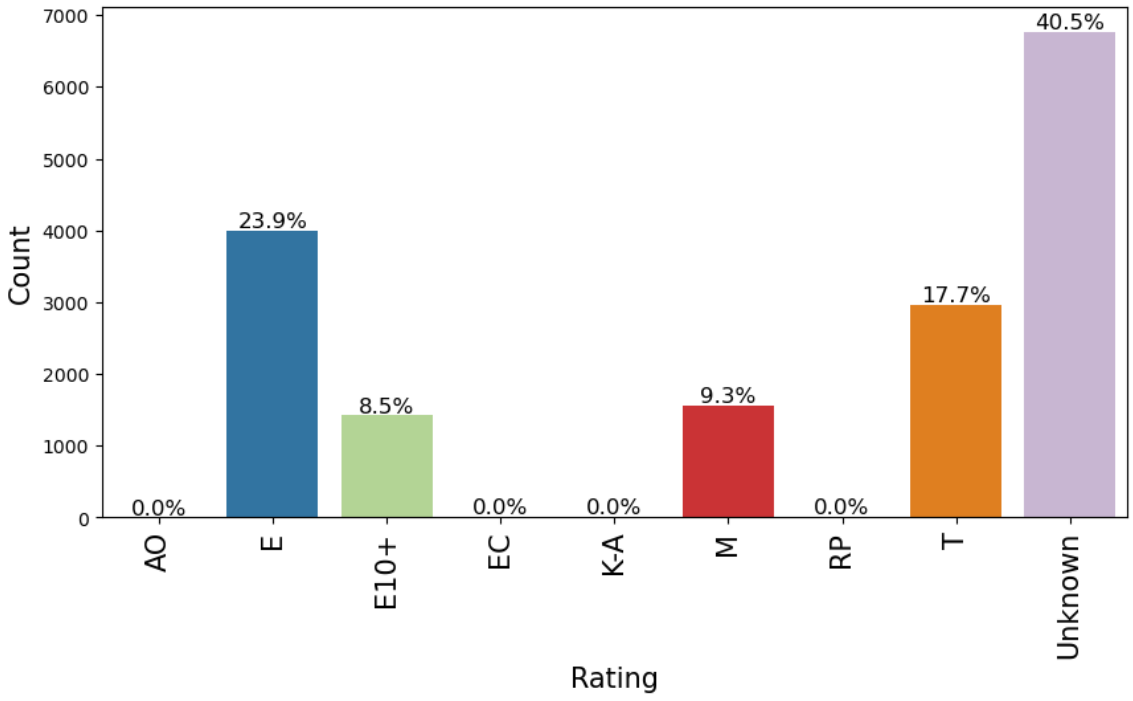
In [44]:

```
1  sns.countplot(data = video_games_model, y = 'Platform', palette = 'deep');
```



In [45]:

```
1  cat_barplot(data=cat_data, feature='Rating', perc= True, n= None )
```



- **The most occurring Rating was 'Unknown' followed by 'E'**

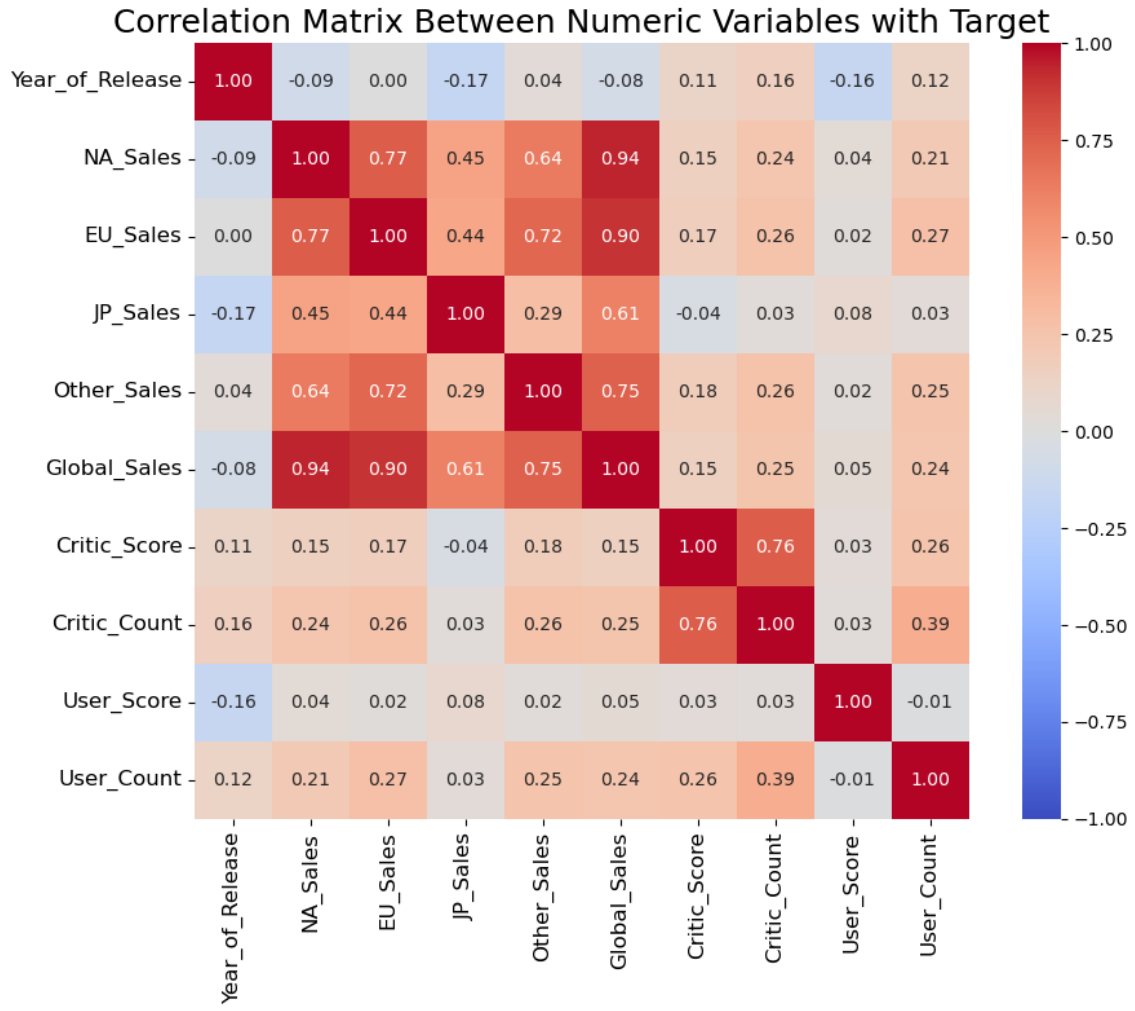
Biavariate Analysis

In [46]:

```
# Select the numeric variables
numeric_feat = video_games_model.select_dtypes(include= 'number')
```

In [47]:

```
# Compute and plot correlation matrix between numeric variables
num_corr = numeric_feat.corr()

fig, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(num_corr, annot=True, fmt='.2f', vmin=-1, vmax=1, cmap='coolwarm', squar

ax.set_title('Correlation Matrix Between Numeric Variables with Target', fontsize=18
ax.set_xticklabels(ax.get_xticklabels(), fontsize=12)
ax.set_yticklabels(ax.get_yticklabels(), fontsize=12)

plt.tight_layout()
plt.show()

```



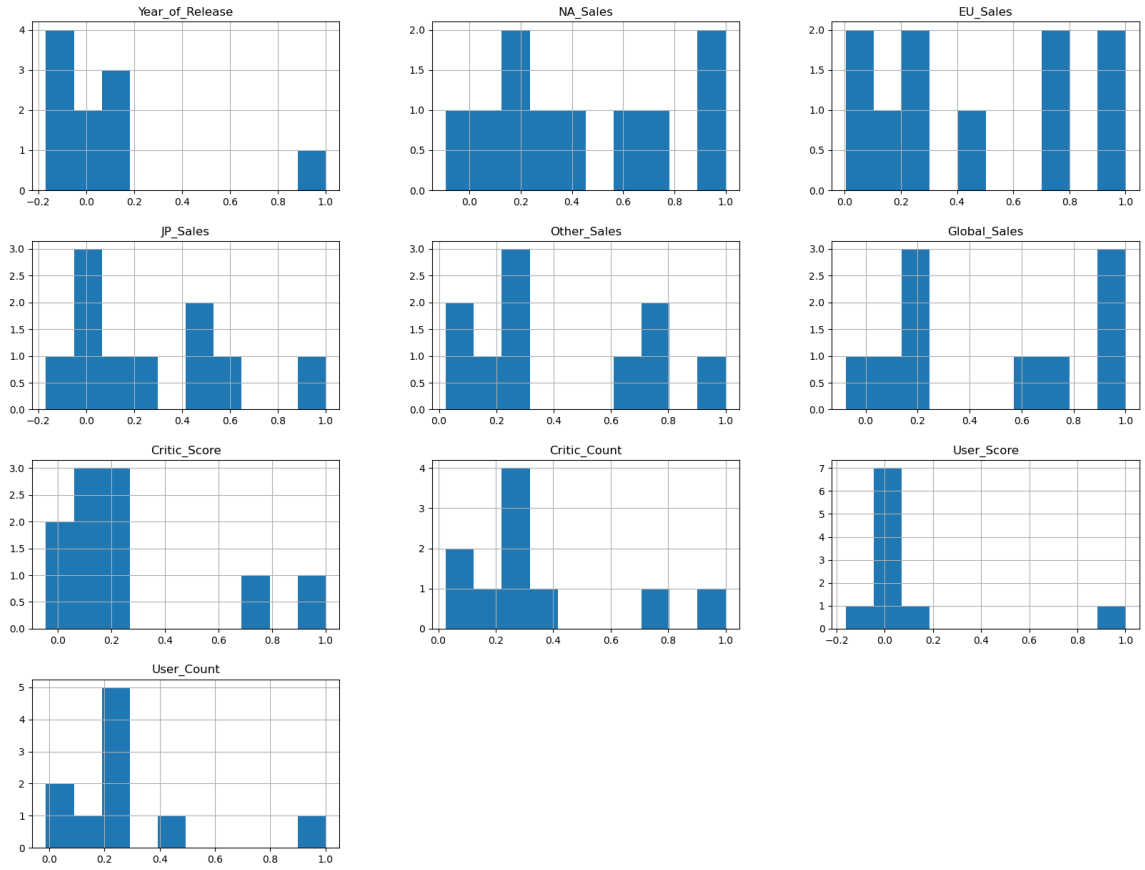
- Sales in NA have significant impact on Global Sales with a correlation of 0.94.
- Sales in EU also have significant impact on Global Sales with a correlation of 0.90.
- Sales in JP have moderate impact on Global Sales with a correlation of 0.61.

- Sales in Other regions have high impact on Global Sales wit 0.75.h a correlation of 0.75.
- Critic Score have low impact on Global Sales with a correlation of 0.15.
- Critic Count have some form of relationship with Global Sales with a correlation of 0.25.

In [48]:

```
# histogram of numeric data

plt.figure(dpi= 120)
num_corr.hist(figsize= (20, 15))
plt.show()
```

```
<Figure size 768x576 with 0 Axes>
```



- **The plots show that the features are not normally distributed**

In [49]:

```
# a quick glance of all the numeric variables
numeric_feat
```

Out[49]:

	Year_of_Release	NA_Sales	EU_Sales	JP_Sales	Other_Sales	Global_Sales	Critic_Sc
0	2006	41.36	28.96	3.77	8.45	82.53	
1	1985	29.08	3.58	6.81	0.77	40.24	
2	2008	15.68	12.76	3.79	3.29	35.52	
3	2009	15.61	10.93	3.28	2.95	32.77	
4	1996	11.27	8.89	10.22	1.00	31.37	
...	...	...	...	...	...	...	
16714	2016	0.00	0.00	0.01	0.00	0.01	
16715	2006	0.00	0.01	0.00	0.00	0.01	
16716	2016	0.00	0.00	0.01	0.00	0.01	
16717	2003	0.01	0.00	0.00	0.00	0.01	
16718	2016	0.00	0.00	0.01	0.00	0.01	

16719 rows × 10 columns

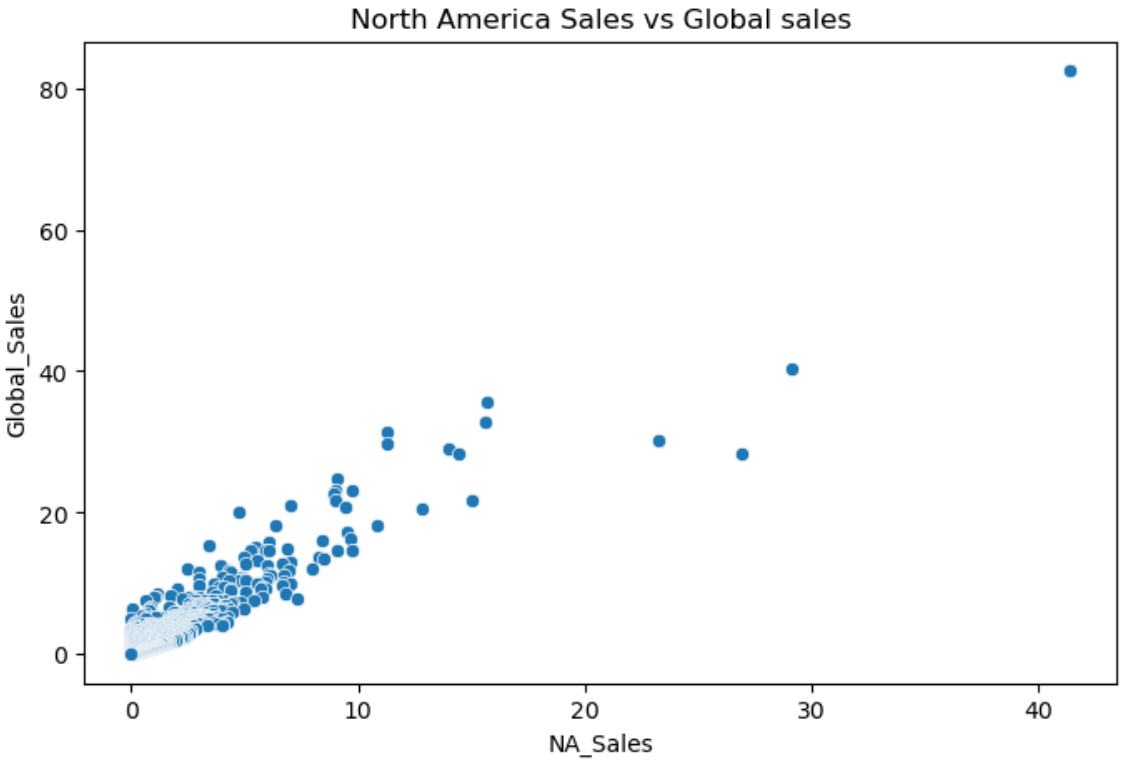
In [50]:

```
# function to create scatter plot between two numeric variables

def num_scatterplot(x, y, title, data = video_games_model):
    plt.figure(figsize= (8,5))
    sns.scatterplot(x = x, y= y, data = video_games_model)
    plt.title(title)
    plt.show()
```

North America Sales vs Global sales

In [51]:

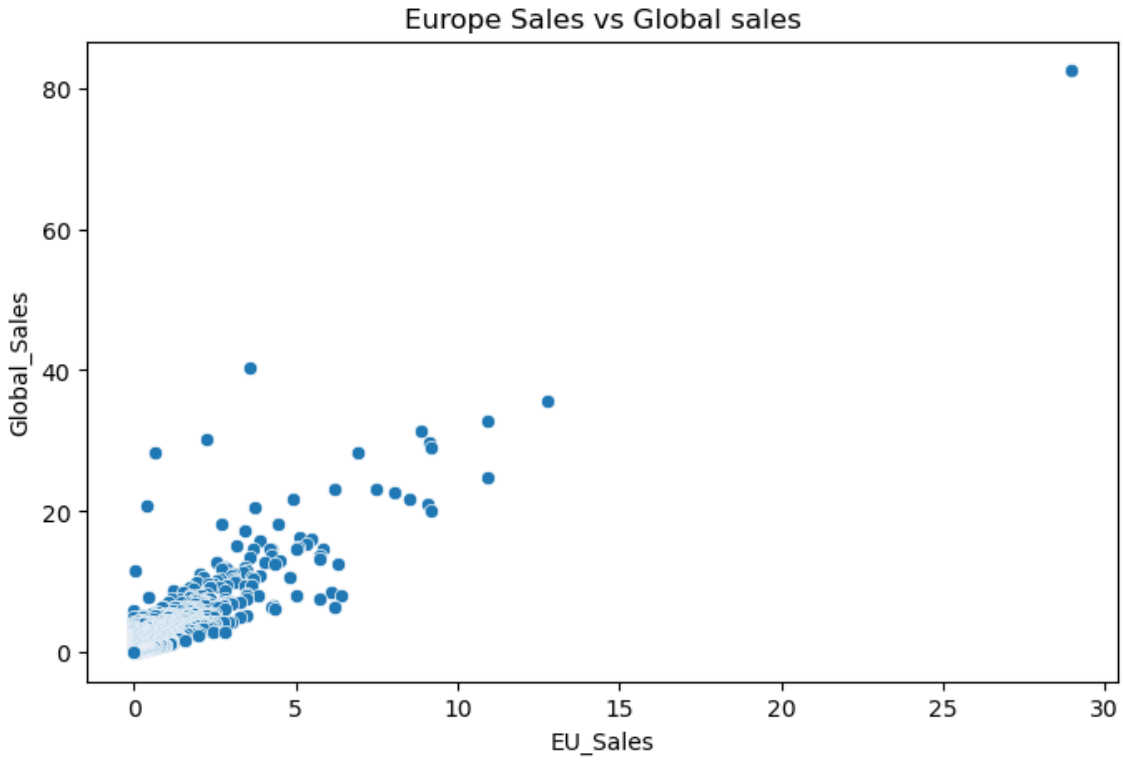
```
# calling the function
num_scatterplot(x = 'NA_Sales', y= 'Global_Sales', title= 'North America Sales vs Gl
```



- **The scatterplot shows that a strong positive relationship exists between North America Sales and Global sales.**
- Global sales increases as the sales in North America increases.

In [52]:

```
# calling the function
num_scatterplot(x = 'EU_Sales', y= 'Global_Sales', title= 'Europe Sales vs Global sa
```

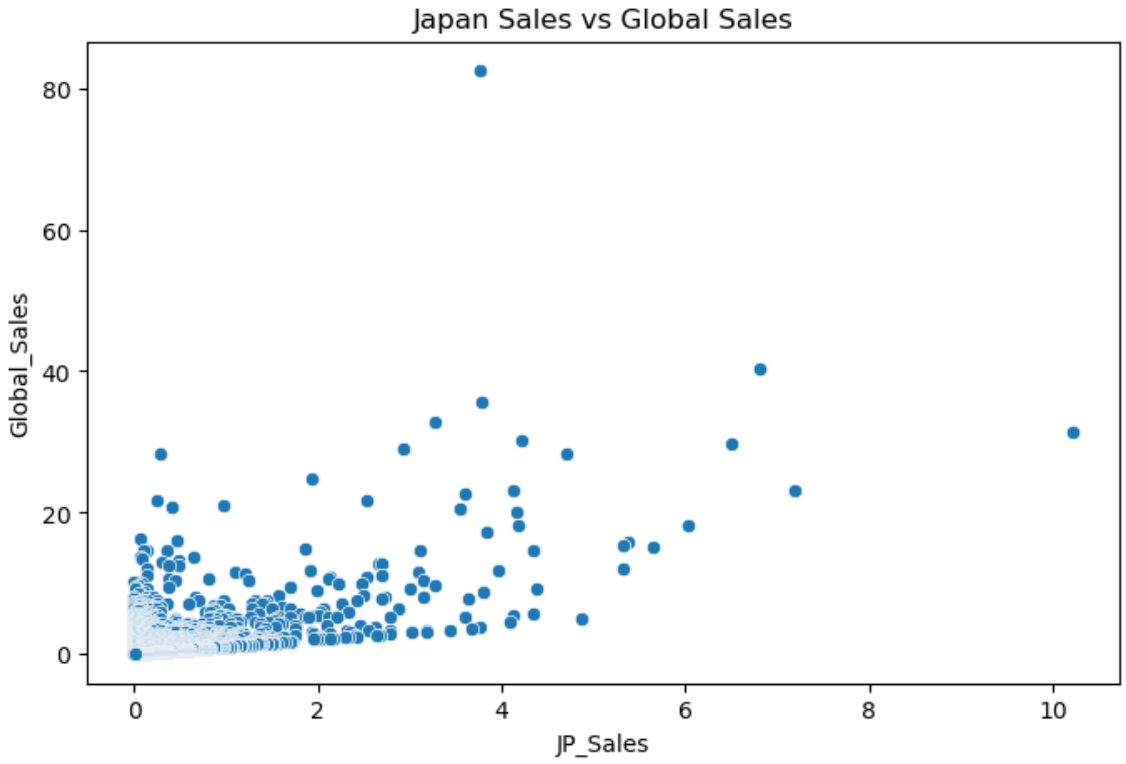


- **The scatterplot shows that a strong positive relationship exists between Europe Sales and Global sales.**

Japan Sales vs Global Sales

In [53]:

```
# calling the scatter plot function
num_scatterplot(x= "JP_Sales", y = "Global_Sales", title= "Japan Sales vs Global Sal
```

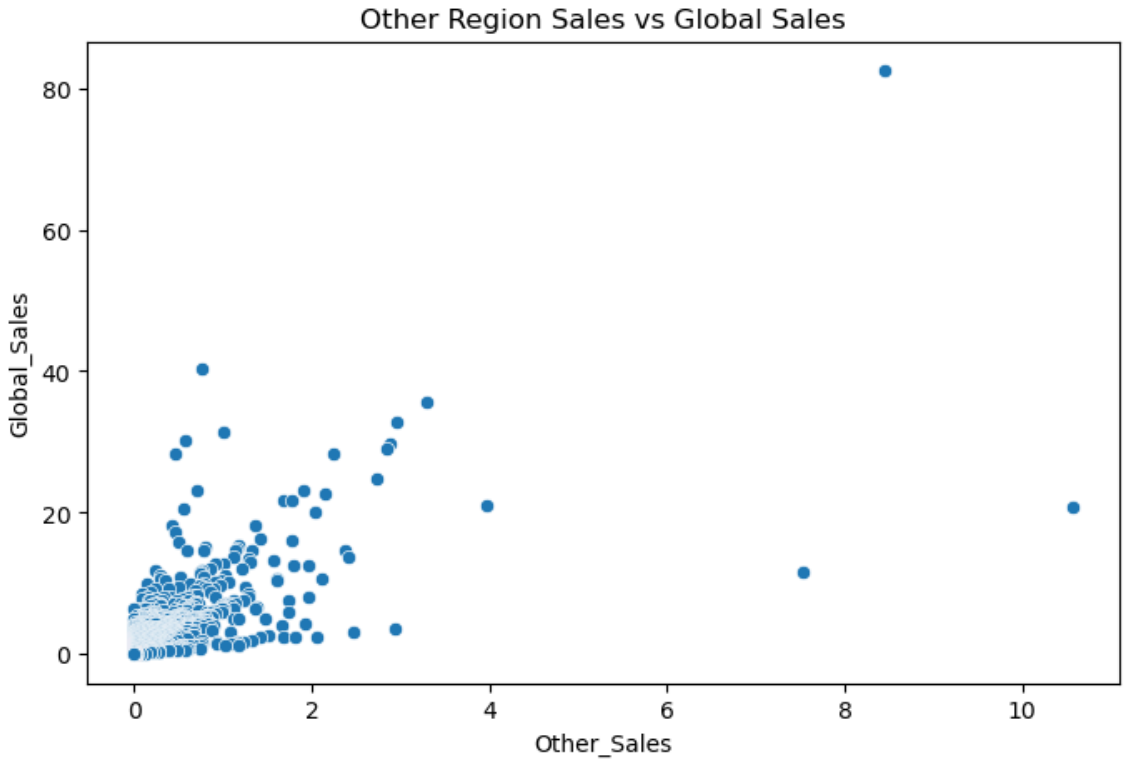


- Japan Sales have a moderately strong relationship with Global Sales even although this could be due to some collinearity issues

Other Region Sales vs Global Sales

In [54]:

```
# calling the scatter plot function to make the Other Region vs Global Sales Scatter
num_scatterplot(x= "Other_Sales", y = "Global_Sales", title= "Other Region Sales vs
```

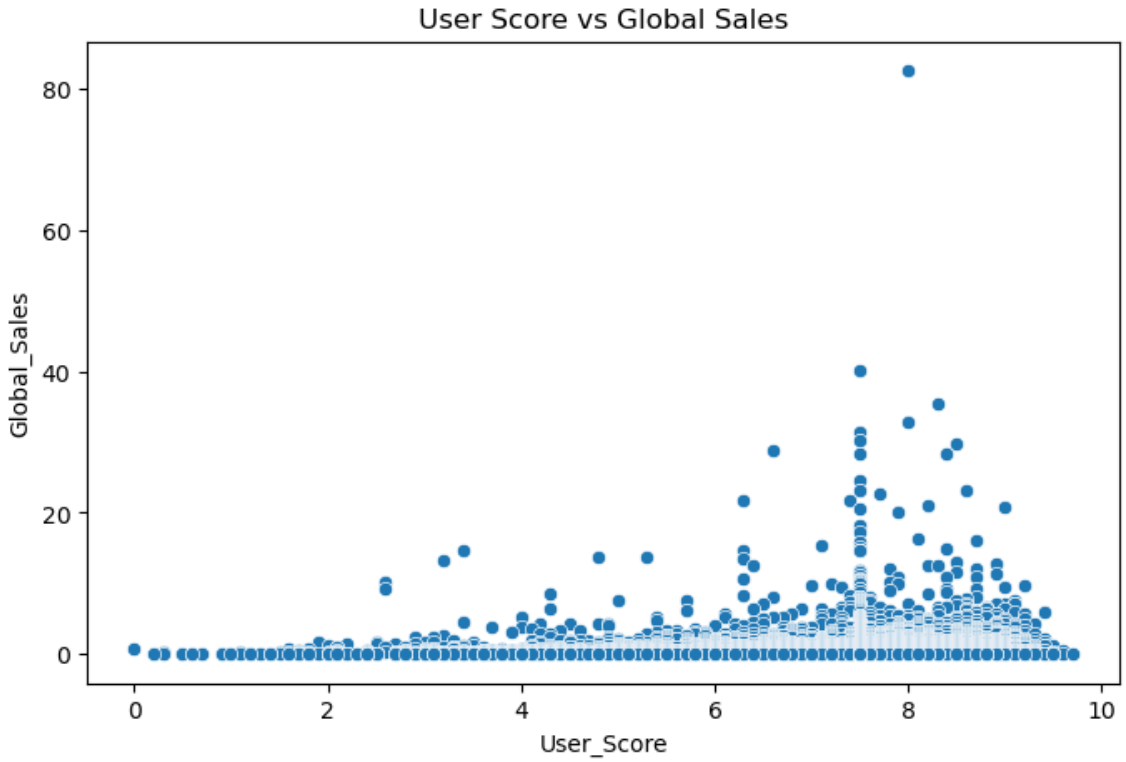


- Other region sales has a positive relationship with Global Sales.

User Score Sales vs Global Sales

In [55]:

```
# Calling the scatter plot function
num_scatterplot(x= "User_Score", y = "Global_Sales", title= "User Score vs Global Sa
```

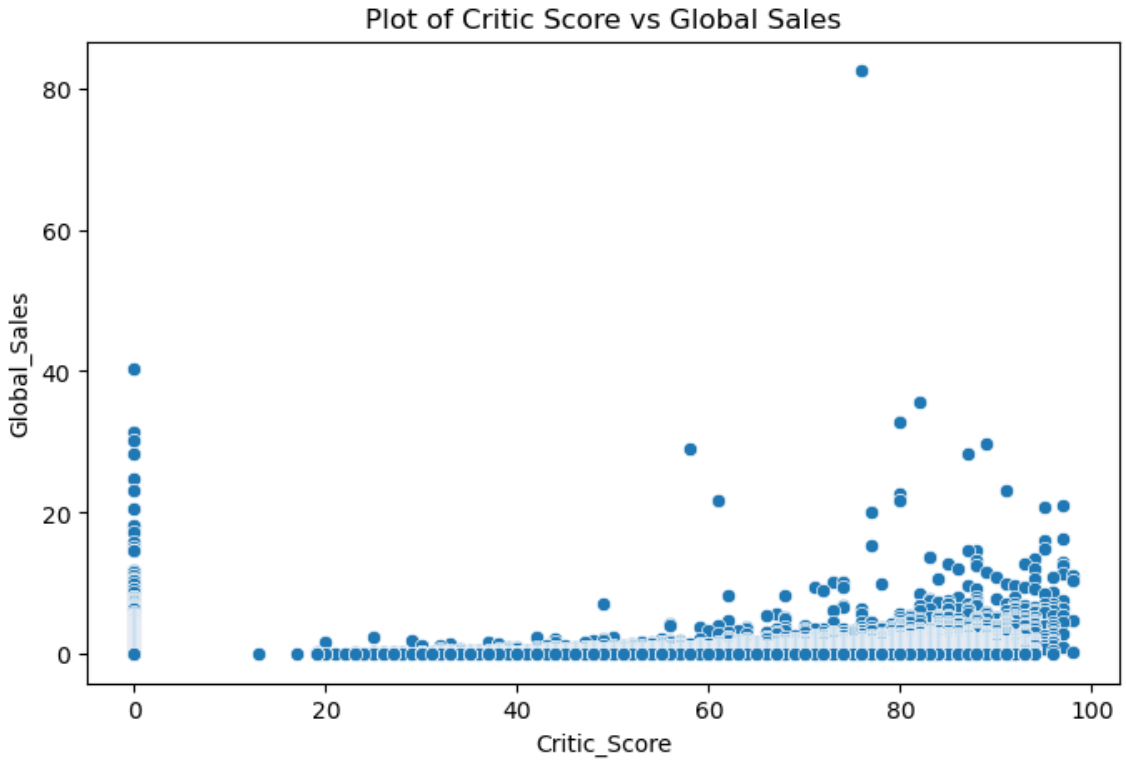


- There is zero to no relationship between User Score and Global Sales

Critic Score vs Global Sales

In [56]:

```
# calling the scatter plot function
num_scatterplot(x= "Critic_Score", y = "Global_Sales", title= "Plot of Critic Score
```

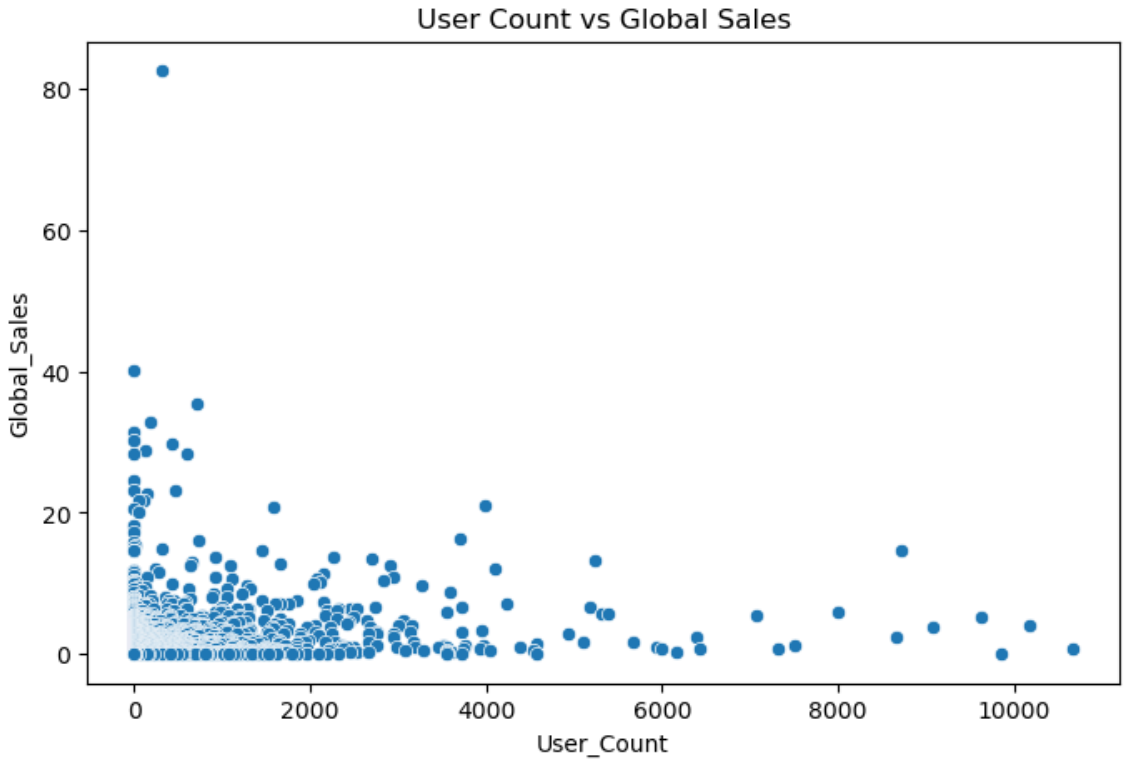


- There is weak positive relationship between Critic Score and Global Sales.

User Count vs Global Sales

In [57]:

```
# calling the scatter plot function
num_scatterplot(x= "User_Count", y = "Global_Sales", title= "User Count vs Global Sa
```



- There is a weak positive relationship between User Count and Global Sales.

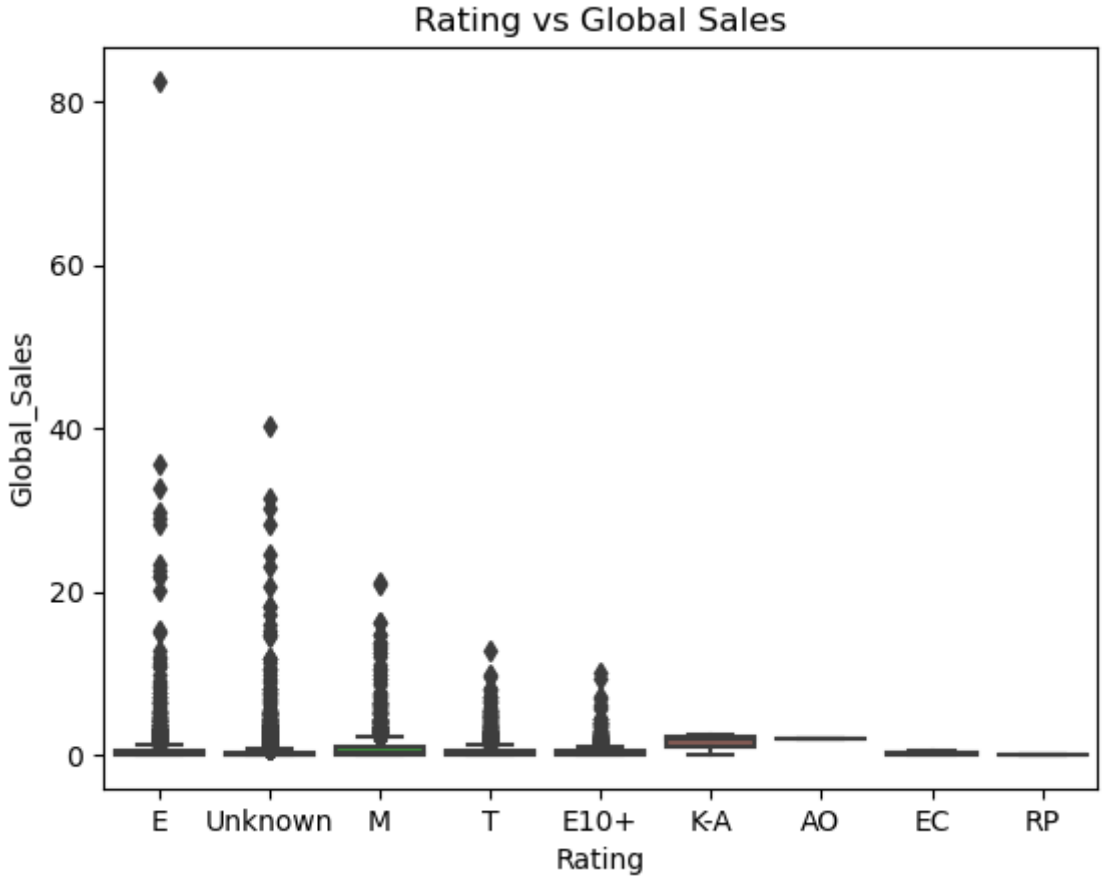
Rating vs Global Sales

In [58]:

```
# using boxplot to show the relationship because 'Rating' is a continuous variable
sns.boxplot( x = "Rating", y = "Global_Sales", data = video_games_model)
plt.title("Rating vs Global Sales")
```

Out[58]:

```
Text(0.5, 1.0, 'Rating vs Global Sales')
```



- Acknowledging the notable outliers, video games with rating as 'E' seems to have more Global Sales

DEVELOP THE MACHINE LEARNING MODEL

- Import required libraries for the regression task

In [130]:

```
from sklearn.preprocessing import StandardScaler # to rescale the numerical data
from sklearn.linear_model import LinearRegression #importing Linear regression
from sklearn.tree import DecisionTreeRegressor
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
from sklearn.model_selection import KFold, cross_val_score, StratifiedKFold, train_t
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error

#importing the needed libraries to convert categorical variables to numeric variable
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import davies_bouldin_score, silhouette_score, calinski_harabas
from sklearn.metrics import adjusted_rand_score, v_measure_score, adjusted_mutual_in
import warnings
warnings.filterwarnings('ignore')
```

Select the model features

In [60]:

```
# Select the model features
feature_models = video_games_model[['NA_Sales','JP_Sales','Other_Sales','EU_Sales','
                                    'User_Count','User_Score']]

target_model = video_games_model['Global_Sales']
```

In [61]:

```
# glance of the features
feature_models
```

Out[61]:

	NA_Sales	JP_Sales	Other_Sales	EU_Sales	Critic_Score	Critic_Count	User_Count
0	41.36	3.77	8.45	28.96	76.0	51.0	322.0
1	29.08	6.81	0.77	3.58	0.0	0.0	0.0
2	15.68	3.79	3.29	12.76	82.0	73.0	709.0
3	15.61	3.28	2.95	10.93	80.0	73.0	192.0
4	11.27	10.22	1.00	8.89	0.0	0.0	0.0
...	...	...	...	...	...	...	...
16714	0.00	0.01	0.00	0.00	0.0	0.0	0.0
16715	0.00	0.00	0.00	0.01	0.0	0.0	0.0
16716	0.00	0.01	0.00	0.00	0.0	0.0	0.0
16717	0.01	0.00	0.00	0.00	0.0	0.0	0.0
16718	0.00	0.01	0.00	0.00	0.0	0.0	0.0

16719 rows × 8 columns

In [62]:

```
# Feature scaling to normalize the data
scaler = StandardScaler()
scaler.fit(feature_models)
scaled_features = scaler.transform(feature_models)
```

In [63]:

```
1 feature_models
```

Out[63]:

	NA_Sales	JP_Sales	Other_Sales	EU_Sales	Critic_Score	Critic_Count	User_Count
0	41.36	3.77	8.45	28.96	76.0	51.0	322.0
1	29.08	6.81	0.77	3.58	0.0	0.0	0.0
2	15.68	3.79	3.29	12.76	82.0	73.0	709.0
3	15.61	3.28	2.95	10.93	80.0	73.0	192.0
4	11.27	10.22	1.00	8.89	0.0	0.0	0.0
...	...	...	...	...	...	...	...
16714	0.00	0.01	0.00	0.00	0.0	0.0	0.0
16715	0.00	0.00	0.00	0.01	0.0	0.0	0.0
16716	0.00	0.01	0.00	0.00	0.0	0.0	0.0
16717	0.01	0.00	0.00	0.00	0.0	0.0	0.0
16718	0.00	0.01	0.00	0.00	0.0	0.0	0.0

16719 rows × 8 columns

In [64]:

```
1 scaled_features =pd.DataFrame(scaled_features, columns = feature_models.columns)
2 scaled_features
```

Out[64]:

	NA_Sales	JP_Sales	Other_Sales	EU_Sales	Critic_Score	Critic_Count	User_Count
0	50.518992	11.956905	45.005218	57.255699	1.184756	2.043467	0.642261
1	35.423530	21.801182	3.870656	6.825337	-0.937162	-0.686837	-0.190461
2	18.951283	12.021669	17.367934	25.066106	1.352276	3.221246	1.643079
3	18.865234	10.370162	15.546873	21.429874	1.296436	3.221246	0.306069
4	13.530193	32.843612	5.102551	17.376370	-0.937162	-0.686837	-0.190461
...	...	...	...	...	...	...	...
16714	-0.323705	-0.218913	-0.253512	-0.288166	-0.937162	-0.686837	-0.190461
16715	-0.323705	-0.251295	-0.253512	-0.268296	-0.937162	-0.686837	-0.190461
16716	-0.323705	-0.218913	-0.253512	-0.288166	-0.937162	-0.686837	-0.190461
16717	-0.311412	-0.251295	-0.253512	-0.288166	-0.937162	-0.686837	-0.190461
16718	-0.323705	-0.218913	-0.253512	-0.288166	-0.937162	-0.686837	-0.190461

16719 rows × 8 columns

Question 2a.

- **Which of the variables in the video game dataset or a combination of them best predicts“global sales” of video games and why?**

In [65]:

```
# Reshape the scaled features for each sales region
scaledX_NA_Sales = np.reshape(scaled_features['NA_Sales'].values, (-1, 1))
scaledX_JP_Sales = np.reshape(scaled_features['JP_Sales'].values, (-1, 1))
scaledX_Other_Sales = np.reshape(scaled_features['Other_Sales'].values, (-1, 1))
scaledX_EU_Sales = np.reshape(scaled_features['EU_Sales'].values, (-1, 1))
```

SIMPLE LINEAR REGRESSION

In [66]:

```
# Creating an instance of the LinearRegression class
lin_reg = LinearRegression()
```

In [67]:

```
kf = KFold(n_splits=5, shuffle=True, random_state=42) #Define the cross validation s

def reg_perf(scaledX, target_model, scaledX_name, linear):

    if linear:
        scaledX = scaled_features[scaledX_name].to_numpy().reshape(-1, 1)
    else:
        scaledX = np.asarray(scaled_features[scaledX_name.split(',')])

    mse_scores = -cross_val_score(lin_reg, scaledX, target_model, scoring='neg_mean_
    mae_scores = -cross_val_score(lin_reg, scaledX, target_model, scoring='neg_mean_
    r2_scores = cross_val_score(lin_reg, scaledX, target_model, scoring='r2', cv=kf)

    print(scaledX_name)
    print(f"Mean Absolute Error score for {scaledX_name}: {mae_scores.mean():.2f}")
    print(f"Mean Squared Error score for {scaledX_name}: {mse_scores.mean():.2f}")
    print(f"Coefficient of determination {scaledX_name}: {r2_scores.mean():.2f}")
    print('\n')

    return
```

In [68]:

```
reg_perf(scaledX_NA_Sales, target_model,"NA_Sales",True)
reg_perf(scaledX_JP_Sales, target_model,"JP_Sales",True)
reg_perf(scaledX_Other_Sales, target_model,"Other_Sales",True)
reg_perf(scaledX_EU_Sales, target_model,"EU_Sales",True)
```

```
NA_Sales
Mean Absolute Error score for NA_Sales: 0.20
Mean Squared Error score for NA_Sales: 0.30
Coefficient of determination NA_Sales: 0.86


JP_Sales
Mean Absolute Error score for JP_Sales: 0.50
Mean Squared Error score for JP_Sales: 1.51
Coefficient of determination JP_Sales: 0.38


Other_Sales
Mean Absolute Error score for Other_Sales: 0.32
Mean Squared Error score for Other_Sales: 1.20
Coefficient of determination Other_Sales: 0.43


EU_Sales
Mean Absolute Error score for EU_Sales: 0.25
Mean Squared Error score for EU_Sales: 0.45
Coefficient of determination EU_Sales: 0.80
```

MULTIPLE LINEAR REGRESSION

In [69]:

```
reg_perf(scaledX= scaledX_NA_Sales, target_model= target_model, scaledX_name= ('Othe
```

```
Other_Sales,EU_Sales,JP_Sales
Mean Absolute Error score for Other_Sales,EU_Sales,JP_Sales: 0.18
Mean Squared Error score for Other_Sales,EU_Sales,JP_Sales: 0.26
Coefficient of determination Other_Sales,EU_Sales,JP_Sales: 0.88
```

In [70]:

```
reg_perf(scaledX= scaledX_EU_Sales, target_model= target_model, scaledX_name= ('NA_S
```

```
NA_Sales,Other_Sales,JP_Sales
Mean Absolute Error score for NA_Sales,Other_Sales,JP_Sales: 0.09
Mean Squared Error score for NA_Sales,Other_Sales,JP_Sales: 0.11
Coefficient of determination NA_Sales,Other_Sales,JP_Sales: 0.95
```

In [71]:

```
1 reg_perf(scaledX= scaledX_JP_Sales, target_model= target_model, scaledX_name= ('NA_S
```

```
NA_Sales,Other_Sales,EU_Sales
Mean Absolute Error score for NA_Sales,Other_Sales,EU_Sales: 0.11
Mean Squared Error score for NA_Sales,Other_Sales,EU_Sales: 0.08
Coefficient of determination NA_Sales,Other_Sales,EU_Sales: 0.96
```

In [72]:

```
1 reg_perf(scaledX= scaledX_Other_Sales, target_model= target_model, scaledX_name= ('N
```

```
NA_Sales,EU_Sales,JP_Sales
Mean Absolute Error score for NA_Sales,EU_Sales,JP_Sales: 0.03
Mean Squared Error score for NA_Sales,EU_Sales,JP_Sales: 0.02
Coefficient of determination NA_Sales,EU_Sales,JP_Sales: 0.99
```

In [73]:

```
1 reg_perf(scaledX= scaledX_EU_Sales, target_model= target_model, scaledX_name= ('NA_S
```

```
NA_Sales,Other_Sales,JP_Sales,EU_Sales
Mean Absolute Error score for NA_Sales,Other_Sales,JP_Sales,EU_Sales: 0.00
Mean Squared Error score for NA_Sales,Other_Sales,JP_Sales,EU_Sales: 0.00
Coefficient of determination NA_Sales,Other_Sales,JP_Sales,EU_Sales: 1.00
```

2a Solution:

- Using Simple Linear Regression, 'NA_Sales' best predicts Global sales as a single variable. Among other individual variables performance, 'NA_Sales' have the highest Coefficient of determination (R-squared) of 0.86, which indicates that only it explains 86% of the variance in global sales. Likewise, the heatmap and scatterplot above demonstrates a strong positive relationship between 'NA_Sales' and 'Global sales'. Futhermore, 'NA_Sales' have the least errors when compared to other individual variables as the Mean Absolute Error (MAE) and Mean Squared Error (MSE) are 0.20 and 0.30 respectively. Since Sales in North America have the smallest values of MAE and MSE, they further suggests it is the best variable for this model (Great Learning Team, 2020) in predicting Global sales. The MAE score of 0.20 suggests the predicted values of 'NA_Sales' are within 0.20 units of the actual values on the average, and the MSE score of 0.30 suggests that, on average, the squared errors have a mean of 0.30.
- Using Multiple Linear Regression, the combination of Sales in North America, Japan, Europe and Other regions have a perfect score for MAE, MSE, and R-squared of 0.00, 0.00 and 1.0 respectively. This could mean that the combination of the four variables best predicts global sales, however, it's important to note that the perfect fit could be as a result of overfitting to the data and may not generalize well to

new data points. Therefore, based on the individual performance of the variables, Sales in North

Great Learning Team (2020) Mean Square Error: Definition, Applications and Examples. GreatLearning. Available online: https://www.mygreatlearning.com/blog/mean-square-error-explained/ (https://www.mygreatlearning.com/blog/mean-square-error-explained/) [Accessed 12 May 2023].

Question 2b.

- **What effect will the number of critics and users as well as their review scores have on the sales of Video games in North America, EU and Japan?**

Building the model

THE REGRESSORS

- Random Forest Regression
- Ridge Regression
- Lasso Regression
- Support Vector Regression
- Decision Tree Regressor

In [74]:

```
### THE REGRESSORS

# Random Forest Regression - used for non-linear regression problems
rf = RandomForestRegressor(n_estimators=100, random_state=42) # using 100 trees and

# initialize Ridge Regression model - used when multicollinearity is present
ridge = Ridge(alpha=1.0) # alpha value set to default 1

# Lasso Regression - used for high-dimensional feature data
lasso = Lasso(alpha=0.1) # alpha value set to 0.1

# Support Vector Regression - used for linear and nonlinear regression
svr = SVR(kernel='linear') # using a linear kernel

# initialize decision tree regressor - used for both classification and regression p
dtree = DecisionTreeRegressor(max_depth=5, random_state=42) # using a maximum depth

```

EU_Sales

In [75]:

```
# Select the columns for the independent variables
variables_x = scaled_features[['Critic_Score', 'Critic_Count', 'User_Count', 'User_S

# Select the column for the target variable
target_Y = video_games_model['EU_Sales']

```

In [76]:

```
# Perform cross-validation and calculate the metrics
mae_scores_rf = -cross_val_score(rf, variables_x, target_Y, cv=kf, scoring='neg_mean
mae_scores_ridge = -cross_val_score(ridge, variables_x, target_Y, cv=kf, scoring='ne
mae_scores_lasso = -cross_val_score(lasso, variables_x, target_Y, cv=kf, scoring='ne
mae_scores_svr = -cross_val_score(svr, variables_x, target_Y, cv=kf, scoring='neg_me
mae_scores_dtree = -cross_val_score(dtree, variables_x, target_Y, cv=kf, scoring='ne

# The Mean Absolute Error - All regressors
print('The Mean Absolute Error for all the Regressors\n')
print('Random Forest Regression MAE for EU_Sales:', round(mae_scores_rf.mean(),2))
print('Ridge Regression MAE for EU_Sales:', round(mae_scores_ridge.mean(),2))
print('Lasso Regression MAE for EU_Sales:', round(mae_scores_lasso.mean(),2))
print('Support Vector Regression MAE for EU_Sales:', round(mae_scores_svr.mean(),2))
print('Decision Tree Regression MAE for EU_Sales:', round(mae_scores_dtree.mean(),2)
```

```
The Mean Absolute Error for all the Regressors

Random Forest Regression MAE for EU_Sales: 0.17
Ridge Regression MAE for EU_Sales: 0.16
Lasso Regression MAE for EU_Sales: 0.18
Support Vector Regression MAE for EU_Sales: 0.16
Decision Tree Regression MAE for EU_Sales: 0.16
```

In [77]:

```
# Perform cross-validation and calculate the metrics
mse_scores_rf = -cross_val_score(rf, variables_x, target_Y, cv=kf, scoring='neg_mean
mse_scores_ridge = -cross_val_score(ridge, variables_x, target_Y, cv=kf, scoring='ne
mse_scores_lasso = -cross_val_score(lasso, variables_x, target_Y, cv=kf, scoring='ne
mse_scores_svr = -cross_val_score(svr, variables_x, target_Y, cv=kf, scoring='neg_me
mse_scores_dtree = -cross_val_score(dtree, variables_x, target_Y, cv=kf, scoring='ne

# The Mean Squared Error - All regressors
print('The Mean Squared Error for all the Regressors\n')
print('Random Forest Regression MSE for EU_Sales:', round(mse_scores_rf.mean(),2))
print('Ridge Regression MSE for EU_Sales:', round(mse_scores_ridge.mean(),2))
print('Lasso Regression MSE for EU_Sales:', round(mse_scores_lasso.mean(),2))
print('Support Vector Regression MSE for EU_Sales:', round(mse_scores_svr.mean(),2))
print('Decision Tree Regression MSE for EU_Sales:', round(mse_scores_dtree.mean(),2)
```

```
The Mean Squared Error for all the Regressors

Random Forest Regression MSE for EU_Sales: 0.23
Ridge Regression MSE for EU_Sales: 0.23
Lasso Regression MSE for EU_Sales: 0.24
Support Vector Regression MSE for EU_Sales: 0.23
Decision Tree Regression MSE for EU_Sales: 0.23
```

In [78]:

```
# Perform cross-validation and calculate the metrics
rmse_scores_rf = np.sqrt(mse_scores_rf)
rmse_scores_ridge = np.sqrt(mse_scores_ridge)
rmse_scores_lasso = np.sqrt(mse_scores_lasso)
rmse_scores_svr = np.sqrt(mse_scores_svr)
rmse_scores_dtree = np.sqrt(mse_scores_dtree)

# The Root Mean Squared Error - All regressors
print('The Root Mean Squared Error for all the Regressors\n')
print('Random Forest Regression RMSE for EU_Sales:', round(rmse_scores_rf.mean(),2))
print('Ridge Regression RMSE for EU_Sales:', round(rmse_scores_ridge.mean(),2))
print('Lasso Regression RMSE for EU_Sales:', round(rmse_scores_lasso.mean(),2))
print('Support Vector Regression RMSE for EU_Sales:', round(rmse_scores_svr.mean(),2
print('Decision Tree Regression RMSE for EU_Sales:', round(rmse_scores_dtree.mean(),
```

```
The Root Mean Squared Error for all the Regressors

Random Forest Regression RMSE for EU_Sales: 0.47
Ridge Regression RMSE for EU_Sales: 0.46
Lasso Regression RMSE for EU_Sales: 0.48
Support Vector Regression RMSE for EU_Sales: 0.47
Decision Tree Regression RMSE for EU_Sales: 0.47
```

In [79]:

```
# Perform cross-validation and calculate the metrics
r2_scores_rf = cross_val_score(rf, variables_x, target_Y, cv=kf, scoring='r2')
r2_scores_ridge = cross_val_score(ridge, variables_x, target_Y, cv=kf, scoring='r2')
r2_scores_lasso = cross_val_score(lasso, variables_x, target_Y, cv=kf, scoring='r2')
r2_scores_svr = cross_val_score(svr, variables_x, target_Y, cv=kf, scoring='r2')
r2_scores_dtree = cross_val_score(dtree, variables_x, target_Y, cv=kf, scoring='r2')

# The coefficient of determination - All regressors
print('The Coefficient of Determination for all the Regressors\n')
print('Random Forest Regression R2 for EU_Sales:', round(r2_scores_rf.mean(),2))
print('Ridge Regression R2 for EU_Sales:', round(r2_scores_ridge.mean(),2))
print('Lasso Regression R2 for EU_Sales:', round(r2_scores_lasso.mean(),2))
print('Support Vector Regression R2 for EU_Sales:', round(r2_scores_svr.mean(),2))
print('Decision Tree Regression R2 for EU_Sales:', round(r2_scores_dtree.mean(),2))
```

```
The Coefficient of Determination for all the Regressors

Random Forest Regression R2 for EU_Sales: 0.08
Ridge Regression R2 for EU_Sales: 0.11
Lasso Regression R2 for EU_Sales: 0.05
Support Vector Regression R2 for EU_Sales: 0.09
Decision Tree Regression R2 for EU_Sales: 0.11
```

SOLUTION REPORT

- The number and scores of Critics as well as that of Users will have no significant impact on the sale of video games in EU. The Coefficient of determination of the different Regressors (Random Forest, Ridge, Lasso, Support Vector and Decision Tree) that were used to evaluate the metrics were all not close to 1. However, Ridge Regression and Decision Tree Regression have the higher R2 values which suggests that they explain a slightly larger proportion of the variance in EU sales by 11%.

NA_Sales

In [80]:

```
# Select the columns for the independent variables
variables_x = scaled_features[['Critic_Score', 'Critic_Count', 'User_Count', 'User_S

# Select the column for the target variable
target_Y = video_games_model['NA_Sales']

```

In [81]:

```
# Perform cross-validation and calculate the metrics for MAE
mae_scores_rf = -cross_val_score(rf, variables_x, target_Y, cv=kf, scoring='neg_mean
mae_scores_ridge = -cross_val_score(ridge, variables_x, target_Y, cv=kf, scoring='ne
mae_scores_lasso = -cross_val_score(lasso, variables_x, target_Y, cv=kf, scoring='ne
mae_scores_svr = -cross_val_score(svr, variables_x, target_Y, cv=kf, scoring='neg_me
mae_scores_dtree = -cross_val_score(dtree, variables_x, target_Y, cv=kf, scoring='ne

# Perform cross-validation and calculate the metrics for MSE
mse_scores_rf = -cross_val_score(rf, variables_x, target_Y, cv=kf, scoring='neg_mean
mse_scores_ridge = -cross_val_score(ridge, variables_x, target_Y, cv=kf, scoring='ne
mse_scores_lasso = -cross_val_score(lasso, variables_x, target_Y, cv=kf, scoring='ne
mse_scores_svr = -cross_val_score(svr, variables_x, target_Y, cv=kf, scoring='neg_me
mse_scores_dtree = -cross_val_score(dtree, variables_x, target_Y, cv=kf, scoring='ne

# Perform cross-validation and calculate the metrics for R2
r2_scores_rf = cross_val_score(rf, variables_x, target_Y, cv=kf, scoring='r2')
r2_scores_ridge = cross_val_score(ridge, variables_x, target_Y, cv=kf, scoring='r2')
r2_scores_lasso = cross_val_score(lasso, variables_x, target_Y, cv=kf, scoring='r2')
r2_scores_svr = cross_val_score(svr, variables_x, target_Y, cv=kf, scoring='r2')
r2_scores_dtree = cross_val_score(dtree, variables_x, target_Y, cv=kf, scoring='r2')
```

In [82]:

```
# The Mean Absolute Error - All regressors
print('The Mean Absolute Error for all the Regressors\n')
print('Random Forest Regression MAE for NA_Sales:', round(mae_scores_rf.mean(),2))
print('Ridge Regression MAE for NA_Sales:', round(mae_scores_ridge.mean(),2))
print('Lasso Regression MAE for NA_Sales:', round(mae_scores_lasso.mean(),2))
print('Support Vector Regression MAE for NA_Sales:', round(mae_scores_svr.mean(),2))
print('Decision Tree Regression MAE for NA_Sales:', round(mae_scores_dtree.mean(),2)

# The Mean Squared Error - All regressors
print('\n')
print('The Mean Squared Error for all the Regressors\n')
print('Random Forest Regression MSE for NA_Sales:', round(mse_scores_rf.mean(),2))
print('Ridge Regression MSE for NA_Sales:', round(mse_scores_ridge.mean(),2))
print('Lasso Regression MSE for NA_Sales:', round(mse_scores_lasso.mean(),2))
print('Support Vector Regression MSE for NA_Sales:', round(mse_scores_svr.mean(),2))
print('Decision Tree Regression MSE for NA_Sales:', round(mse_scores_dtree.mean(),2)

# The coefficient of determination - All regressors
print('\n')
print('The Coefficient of Determination for all the Regressors\n')
print('Random Forest Regression R2 for NA_Sales:', round(r2_scores_rf.mean(),2))
print('Ridge Regression R2 for NA_Sales:', round(r2_scores_ridge.mean(),2))
print('Lasso Regression R2 for NA_Sales:', round(r2_scores_lasso.mean(),2))
print('Support Vector Regression R2 for NA_Sales:', round(r2_scores_svr.mean(),2))
print('Decision Tree Regression R2 for NA_Sales:', round(r2_scores_dtree.mean(),2))
```

```
The Mean Absolute Error for all the Regressors

Random Forest Regression MAE for NA_Sales: 0.28
Ridge Regression MAE for NA_Sales: 0.28
Lasso Regression MAE for NA_Sales: 0.29
Support Vector Regression MAE for NA_Sales: 0.24
Decision Tree Regression MAE for NA_Sales: 0.27


The Mean Squared Error for all the Regressors

Random Forest Regression MSE for NA_Sales: 0.62
Ridge Regression MSE for NA_Sales: 0.61
Lasso Regression MSE for NA_Sales: 0.63
Support Vector Regression MSE for NA_Sales: 0.64
Decision Tree Regression MSE for NA_Sales: 0.6


The Coefficient of Determination for all the Regressors

Random Forest Regression R2 for NA_Sales: 0.07
Ridge Regression R2 for NA_Sales: 0.09
Lasso Regression R2 for NA_Sales: 0.06
Support Vector Regression R2 for NA_Sales: 0.04
Decision Tree Regression R2 for NA_Sales: 0.12
```

SOLUTION REPORT

- The MAE, MSE and R2 metrics evaluation above show that critic count and critic score as well as user count and score have a moderate effect on the video games sales in North America. Support Vector Regression performs relatively better in terms of Mean Absolute Error, which is 0.24, while Decision

Tree Regression shows slightly better performance in terms of Mean Squared Error and Coefficient of Determination of 0.6 and 0.12, respectively. The 12% R2 score by Decision Tree Regressor captures a slightly larger proportion of the variability in NA_Sales compared to the other regressors.

JP_Sales

In [83]:

```
# Select the columns for the independent variables
variables_x = scaled_features[['Critic_Score', 'Critic_Count', 'User_Count', 'User_S

# Select the column for the target variable
target_Y = video_games_model['JP_Sales']

```

In [84]:

```
# Perform cross-validation and calculate the metrics for MAE
mae_scores_rf = -cross_val_score(rf, variables_x, target_Y, cv=kf, scoring='neg_mean
mae_scores_ridge = -cross_val_score(ridge, variables_x, target_Y, cv=kf, scoring='ne
mae_scores_lasso = -cross_val_score(lasso, variables_x, target_Y, cv=kf, scoring='ne
mae_scores_svr = -cross_val_score(svr, variables_x, target_Y, cv=kf, scoring='neg_me
mae_scores_dtree = -cross_val_score(dtree, variables_x, target_Y, cv=kf, scoring='ne

# Perform cross-validation and calculate the metrics for MSE
mse_scores_rf = -cross_val_score(rf, variables_x, target_Y, cv=kf, scoring='neg_mean
mse_scores_ridge = -cross_val_score(ridge, variables_x, target_Y, cv=kf, scoring='ne
mse_scores_lasso = -cross_val_score(lasso, variables_x, target_Y, cv=kf, scoring='ne
mse_scores_svr = -cross_val_score(svr, variables_x, target_Y, cv=kf, scoring='neg_me
mse_scores_dtree = -cross_val_score(dtree, variables_x, target_Y, cv=kf, scoring='ne

# Perform cross-validation and calculate the metrics for R2
r2_scores_rf = cross_val_score(rf, variables_x, target_Y, cv=kf, scoring='r2')
r2_scores_ridge = cross_val_score(ridge, variables_x, target_Y, cv=kf, scoring='r2')
r2_scores_lasso = cross_val_score(lasso, variables_x, target_Y, cv=kf, scoring='r2')
r2_scores_svr = cross_val_score(svr, variables_x, target_Y, cv=kf, scoring='r2')
r2_scores_dtree = cross_val_score(dtree, variables_x, target_Y, cv=kf, scoring='r2')
```

In [85]:

```
# The Mean Absolute Error - All regressors
print('The Mean Absolute Error for all the Regressors\n')
print('Random Forest Regression MAE for JP_Sales:', round(mae_scores_rf.mean(),2))
print('Ridge Regression MAE for JP_Sales:', round(mae_scores_ridge.mean(),2))
print('Lasso Regression MAE for JP_Sales:', round(mae_scores_lasso.mean(),2))
print('Support Vector Regression MAE for JP_Sales:', round(mae_scores_svr.mean(),2))
print('Decision Tree Regression MAE for JP_Sales:', round(mae_scores_dtree.mean(),2)

# The Mean Squared Error - All regressors
print('\n')
print('The Mean Squared Error for all the Regressors\n')
print('Random Forest Regression MSE for JP_Sales:', round(mse_scores_rf.mean(),2))
print('Ridge Regression MSE for JP_Sales:', round(mse_scores_ridge.mean(),2))
print('Lasso Regression MSE for JP_Sales:', round(mse_scores_lasso.mean(),2))
print('Support Vector Regression MSE for JP_Sales:', round(mse_scores_svr.mean(),2))
print('Decision Tree Regression MSE for JP_Sales:', round(mse_scores_dtree.mean(),2)

# The coefficient of determination - All regressors
print('\n')
print('The Coefficient of Determination for all the Regressors\n')
print('Random Forest Regression R2 for JP_Sales:', round(r2_scores_rf.mean(),2))
print('Ridge Regression R2 for JP_Sales:', round(r2_scores_ridge.mean(),2))
print('Lasso Regression R2 for JP_Sales:', round(r2_scores_lasso.mean(),2))
print('Support Vector Regression R2 for JP_Sales:', round(r2_scores_svr.mean(),2))
print('Decision Tree Regression R2 for JP_Sales:', round(r2_scores_dtree.mean(),2))
```

```
The Mean Absolute Error for all the Regressors

Random Forest Regression MAE for JP_Sales: 0.11
Ridge Regression MAE for JP_Sales: 0.11
Lasso Regression MAE for JP_Sales: 0.12
Support Vector Regression MAE for JP_Sales: 0.13
Decision Tree Regression MAE for JP_Sales: 0.11


The Mean Squared Error for all the Regressors

Random Forest Regression MSE for JP_Sales: 0.1
Ridge Regression MSE for JP_Sales: 0.09
Lasso Regression MSE for JP_Sales: 0.1
Support Vector Regression MSE for JP_Sales: 0.1
Decision Tree Regression MSE for JP_Sales: 0.09


The Coefficient of Determination for all the Regressors

Random Forest Regression R2 for JP_Sales: 0.0
Ridge Regression R2 for JP_Sales: 0.02
Lasso Regression R2 for JP_Sales: -0.0
Support Vector Regression R2 for JP_Sales: -0.01
Decision Tree Regression R2 for JP_Sales: 0.01
```

SOLUTION REPORT

- From the above metrics, the number of Critics and Users, as well as their review scores, have weak to no effect on predicting the sales of video games in Japan. Considering the MAE, MSE, and R2, Ridge Regression and Decision Tree Regression performed slightly better than the other regressors with their

lowest Mean Absolute Error and Mean Squared Error of 0.11, as well as, the marginally higher R-squared of 0.02 and 0.01 respectively. Although Random Forest Regression also have the same MAE of 0.11, it did not perform well in MSE and Coefficient of Determination.

- Overall, as the lower value of MAE, MSE, and RMSE implies higher accuracy of a regression model, while a higher value of R square is considered desirable (Chugh, 2020), Decision Tree Regression appears to be the best regressor for predicting sales of Video games in North America, EU and Japan against the stated features because of its low MAE and MSE and its high R2 across the regions.

Chugh, A. (2020) MAE, MSE, RMSE, Coefficient of Determination, Adjusted R Squared — Which Metric is Better? Medium. Available online: https://medium.com/analytics-vidhya/mae-mse-rmse-coefficient-of-determination-adjusted-r-squared-which-metric-is-better-cd0326a5697e (https://medium.com/analytics-vidhya/mae-mse-rmse-coefficient-of-determination-adjusted-r-squared-which-metric-is-better-cd0326a5697e) [Accessed 12 May 2023].

Question 2c.

- **What propelled the choice of your regressor for this task? Aptly discuss with quantitative reasons!**

- Decision Tree Regression as a machine learning algorithm explores complex relationships and outliers in the data (Pedregosa, et al., 2011). Although decision trees are prone to overfitting, I evaluated the performance of the model and compared it to other regressors before finalizing my choice. It is the overall best regressor as it performed generally well in all the evaluation metrics across the regions.
  - For Sales in EU, the MAE for Decision Tree is 0.16 (among the lowest MAE), while the MSE is 0.23 (one of the lowest MSE) and R-squared of 0.11 (one of the highest value) making it the best model when compared to other regressors.
  - For Sales in North America, the MAE for Decision Tree is 0.27 (one of the lowest MAE), while the MSE is 0.6 (the lowest MSE) and R-squared of 0.12 (the highest R2) making it the best model when compared to other regressors.
  - For Sales in Japan, the MAE for Decision Tree is 0.11 (one of the lowest MAE), while the MSE is 0.09 (one of the lowest MSE) and R-squared of 0.01 (one of the highest value) making it the best model when compared to other regressors.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M. and Prettenhofer, P. (2011) About us — scikit-learn 0.23.1 documentation. Scikit-learn.org. Available online: https://scikit-learn.org/stable/about.html#citing-scikit-learn (https://scikit-learn.org/stable/about.html#citing-scikit-learn) [Accessed 12 May 2023].

Question 2d.

- **Use all the relevant categorical variables in the Video Game Dataset as the target variable at each instance and determine which of the variables performed best in classifying the dataset. Explain your findings.**

In [86]:

```
# Selects 5 random rows from the categorical dataframe
cat_data.sample(5)
```

Out[86]:

	Name	Platform	Genre	Publisher	Developer	Rating
5104	Tiger Woods PGA Tour 14	PS3	Sports	Electronic Arts	EA Tiburon	E
11028	Pro Evolution Soccer 2012	PC	Action	Konami Digital Entertainment	Konami	Unknown
771	Super Mario RPG: Legend of the Seven Stars	SNES	Role-Playing	Nintendo	Ubisoft	Unknown
9765	Curious George	PS2	Action	Namco Bandai Games	Monkey Bar Games	E
270	Fallout 4	XOne	Role-Playing	Bethesda Softworks	Bethesda Game Studios	M

In [87]:

```
video_games_model['Name'].nunique()
```

Out[87]:

```
11562
```

In [88]:

```
video_games_model['Platform'].nunique()
```

Out[88]:

```
31
```

In [89]:

```
video_games_model['Genre'].nunique()
```

Out[89]:

```
12
```

In [90]:

```
video_games_model['Publisher'].nunique()
```

Out[90]:

```
581
```

In [91]:

```
1 video_games_model['Developer'].nunique()
```

Out[91]:

1696

In [92]:

```
1 video_games_model['Rating'].nunique()
```

Out[92]:

9

- I consider Platform, Genre and Rating as the relevant categorical variables which may perform best in classifying the dataset.

In [93]:

```
1 object_float_df = video_games_model.select_dtypes(include = "float64")
```

Platform as Target

In [94]:

```
1 video_games_model['Platform'].value_counts()
```

Out[94]:

```
PS2     2161
DS      2152
PS3     1331
Wii     1320
X360    1262
PSP     1209
PS      1197
PC       974
XB       824
GBA      822
GC       556
3DS      520
PSV      432
PS4      393
N64      319
XOne     247
SNES     239
SAT      173
WiiU     147
2600     133
NES       98
GB        98
DC        52
GEN       29
NG        12
SCD        6
WS         6
3DO        3
TG16       2
GG         1
PCFX       1
Name: Platform, dtype: int64
```

In [95]:

```
1 # Replacing the Platforms with less than 4 values with the mode of the column
2 video_games_model['Platform'] = video_games_model['Platform'].replace(['TG16', 'GG',
```

In [96]:

```
# Confirming the changes
video_games_model['Platform'].value_counts()
```

Out[96]:

```
PS2     2168
DS      2152
PS3     1331
Wii     1320
X360    1262
PSP     1209
PS      1197
PC       974
XB       824
GBA      822
GC       556
3DS      520
PSV      432
PS4      393
N64      319
XOne     247
SNES     239
SAT      173
WiiU     147
2600     133
GB        98
NES       98
DC        52
GEN       29
NG        12
SCD        6
WS         6
Name: Platform, dtype: int64
```

In [97]:

```
# create an instance of OneHotEncoder
onehot_encoder = OneHotEncoder(sparse=False)

# Applying One-hot encoding on 'Rating' and 'Genre' column
rating_genre_encoded = onehot_encoder.fit_transform(video_games_model[['Rating', 'Ge

# Getting feature names for 'Platform' column
features = onehot_encoder.get_feature_names_out(['Rating', 'Genre'])

# Creating a DataFrame with the one-hot encoded 'Rating' and 'Genre' column
rating_genre_onehot_encoded_df = pd.DataFrame(data=rating_genre_encoded, columns=fea

# Creating an instance of Labelencoder
label_encoder = LabelEncoder()

# Converting the target variable to numerical using Lable encoding.
platform_labeled = label_encoder.fit_transform(video_games_model['Platform'])

# Creating a DataFrame with the numerical'Platform' column
platform_labeled_df = pd.DataFrame(platform_labeled, columns=['Platform'])

# Combining the one-hot encoded 'Rating' and 'Genre' and numerical 'Platform' DataFr
data_class_plat = pd.concat([rating_genre_onehot_encoded_df, platform_labeled_df, ob
data_class_plat
```

Out[97]:

	Rating_AO	Rating_E	Rating_E10+	Rating_EC	Rating_K-A	Rating_M	Rating_RP	Rati
0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
3	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
...	...	...	...	...	...	...	...	
16714	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
16715	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
16716	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
16717	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
16718	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

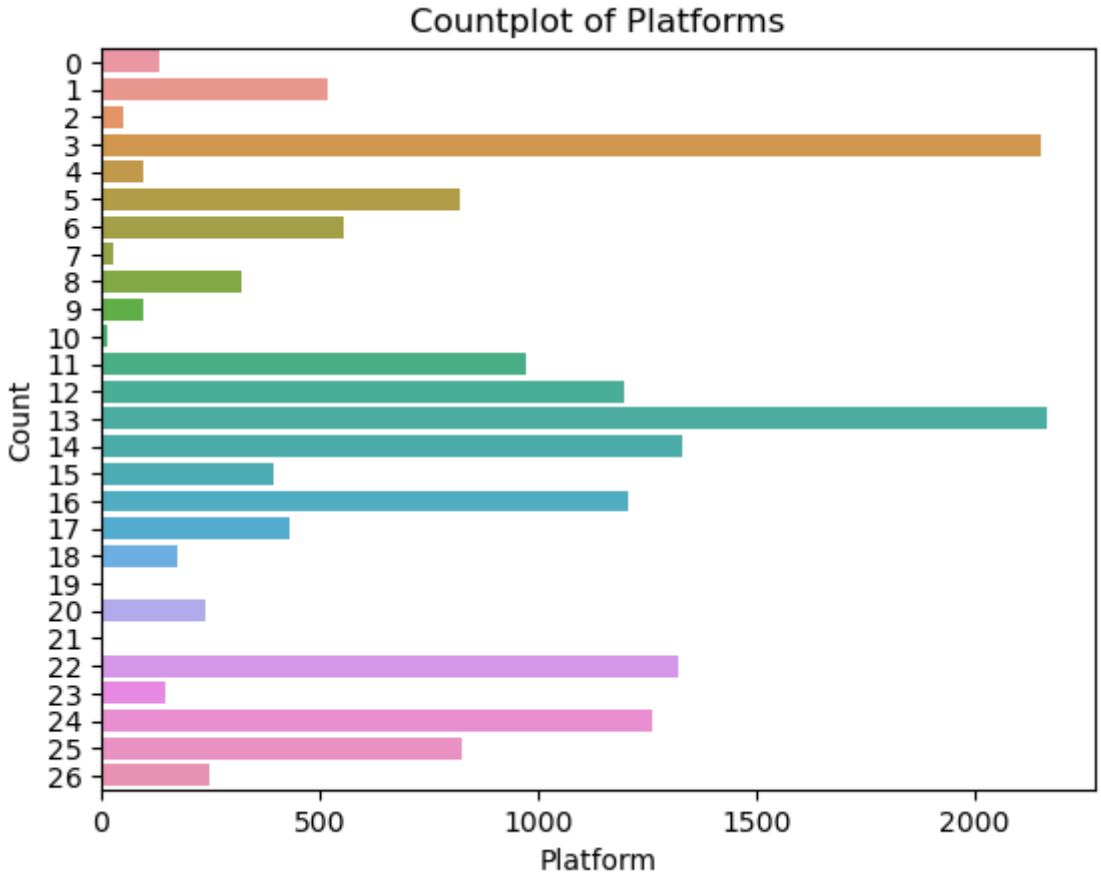
16719 rows × 31 columns

In [98]:

```
# Split data into input(X) and output(Y) variables
X = data_class_plat.drop(['Platform'],axis=1)
y= data_class_plat['Platform']
```

In [99]:

```
# Plotting the target(Platform) variable
sns.countplot(y='Platform', data=data_class_plat)

plt.title('Countplot of Platforms')
plt.xlabel('Platform')
plt.ylabel('Count')

#Show the plot
plt.show()
```



- The plot is not evenly distributed.

APPLYING SMOTE

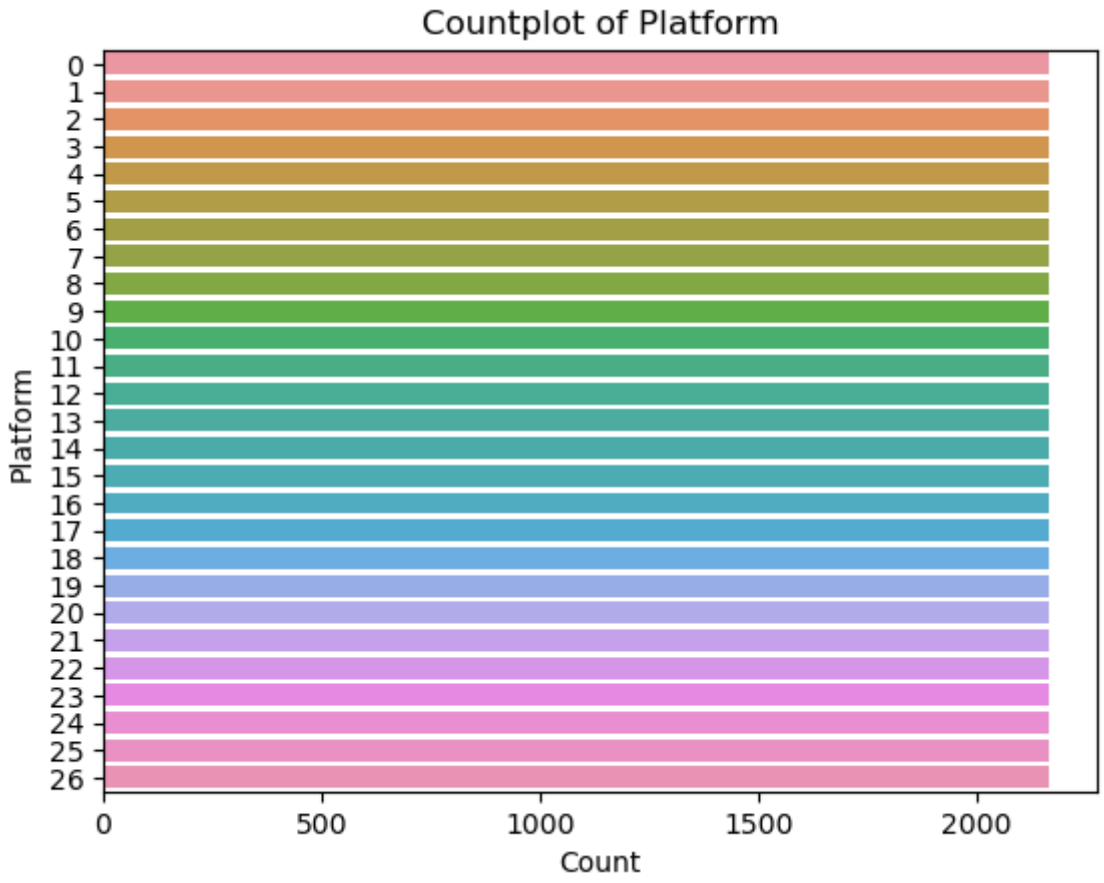
- Since our target class label data is not equally distributed, Synthetic Minority Oversampling Technique (SMOTE) will assist us to balance it.

In [100]:

```
# import SMOTE
from imblearn.over_sampling import SMOTE
sm = SMOTE(random_state=42, k_neighbors = 4) # The object is created
# Apply SMOTE to resample the dataset
X_res, y_res = sm.fit_resample(X, y) # The object is applied
# Reassigning the balanced dataset to X,y
X, y = X_res, y_res

# Plot of the dataset
ax = sns.countplot(y=y, data = data_class_plat )
# set the labels for the axes and the title
ax.set_xlabel('Count')
ax.set_ylabel('Platform')
ax.set_title('Countplot of Platform')
```

Out[100]:

```
Text(0.5, 1.0, 'Countplot of Platform')
```



- Now it has been distributed evenly.

In [101]:

```
# Initialize the classifier
clf = DecisionTreeClassifier()

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Perform cross_validation

scores = cross_val_score(clf,X,y,cv=cv)

# Fit classifier on the dataset.

clf.fit(X, y)

# Make predictions on the test data.

y_pred = clf.predict(X)
```

In [102]:

```
y_pred
```

Out[102]:

```
array([22,  9, 22, ..., 26, 26, 26])
```

In [103]:

```
# Prints the mean and standard deviation of scores
print('Cross-validation scores:', scores)
print('Mean score:', scores.mean())
print('Standard deviation:', scores.std())
print('Train score:', scores.mean() + 2*scores.std())
```

```
Cross-validation scores: [0.66655278 0.67660374 0.67856838 0.67122235 0.66
942855]
Mean score: 0.6724751592588294
Standard deviation: 0.0044743360470889774
Train score: 0.6814238313530073
```

In [104]:

```
# Prints the Classification report
print('Classification Report:')
print(classification_report(y, y_pred))
```

```
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2168
           1       0.83      0.88      0.85      2168
           2       0.99      0.99      0.99      2168
           3       0.81      0.81      0.81      2168
           4       0.99      0.99      0.99      2168
           5       0.98      0.98      0.98      2168
           6       0.99      0.99      0.99      2168
           7       0.98      0.99      0.99      2168
           8       0.99      0.98      0.99      2168
           9       1.00      1.00      1.00      2168
          10       1.00      1.00      1.00      2168
          11       0.98      0.98      0.98      2168
          12       0.98      0.98      0.98      2168
          13       0.93      0.83      0.88      2168
          14       0.97      0.88      0.92      2168
          15       1.00      0.96      0.98      2168
          16       0.82      0.70      0.75      2168
          17       0.64      0.91      0.75      2168
          18       0.92      0.95      0.93      2168
          19       1.00      1.00      1.00      2168
          20       0.96      0.93      0.95      2168
          21       1.00      1.00      1.00      2168
          22       0.97      0.90      0.93      2168
          23       0.97      0.99      0.98      2168
          24       0.99      0.95      0.97      2168
          25       0.99      0.99      0.99      2168
          26       1.00      0.99      0.99      2168

    accuracy                           0.95     58536
   macro avg       0.95      0.95      0.95     58536
weighted avg       0.95      0.95      0.95     58536
```

In [105]:

```
# Prints the confusion Matrix
print('Confusion Matrix:')
print(confusion_matrix(y, y_pred))
```

```
Confusion Matrix:
[[2168    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0]
 [   0 1898    0   18    0    1    0   16    0    0    0    2    0    6
     7    0   24  179    8    0    3    0    2    4    0    0    0]
 [   0    2 2139    0    0    0    0    0    0    0    0    0    0    3
     0    0    1    2   19    0    1    0    0    0    1    0    0]
 [   1   78    6 1754    5    6    1    4    3    0    0   12    2   17
     2    0   70  124   18    0   19    0   30    9    1    3    3]
 [   0    2    0    4 2154    0    0    1    0    0    0    0    0    0
     1    0    2    0    0    0    4    0    0    0    0    0    0]
 [   0    3    1   12    2 2117    0    4    1    0    0    0    6    0
     1    0    7    6    4    0    0    0    0    0    0    4    0]
 [   0    1    0    3    0   10 2145    0    2    0    0    0    1    3
     1    0    1    0    1    0    0    0    0    0    0    0    0]
 [   0    0    0    1    1    0    1 2156    0    0    0    0    0    1
     0    0    2    3    2    0    0    0    0    1    0    0    0]
 [   1    0    1    6    1    7    7    0 2134    0    0    0    1    2
     2    0    0    0    2    0    1    0    1    0    0    2    0]
 [   0    0    0    2    3    0    0    0    0 2161    0    0    0    0
     0    0    0    0    1    0    1    0    0    0    0    0    0]
 [   0    1    2    1    0    0    0    0    0    0 2158    0    0    2
     0    0    2    0    2    0    0    0    0    0    0    0    0]
 [   0    8    0   16    0    2    1    0    0    0    0 2131    2    0
     0    2    0    0    0    0    0    0    5    0    1    0    0]
 [   0    5    0    7    0    6    0    0    1    0    0    1 2128    3
     2    0    3    5    1    0    1    0    0    3    0    2    0]
 [   0   43    4   55    2    2    0    3    2    0    1    0   13 1803
     4    0   44  135   32    0   12    0    0    8    5    0    0]
 [   0   30    1   40    1    0    0    2    2    0    0    6    2   18
  1900    1   33  101   10    0    7    0    1    9    2    2    0]
 [   0   10    0    4    0    0    0    0    0    0    0    2    0    4
     2 2078   10   51    2    0    2    0    1    1    0    0    1]
 [   0   75    4   62    4    0    1    6    1    0    0    1    0   28
     7    1 1514  394   34    0   18    0    7   11    0    0    0]
 [   0   68    6   21    0    1    0    1    1    0    0    2    0    3
     9    1   72 1974    6    0    1    0    1    1    0    0    0]
 [   0   19    1   14    3    1    0    1    0    0    0    0    0   15
     3    0   24   27 2053    0    4    0    1    1    1    0    0]
 [   0    0    0    1    0    0    0    0    0    0    0    0    0    0
     0    0    0    1    1 2164    1    0    0    0    0    0    0]
 [   0   15    1   19    3    0    0   12    6    1    0    0    0   13
     6    1   20   23   15    0 2025    0    0    8    0    0    0]
 [   0    2    0    1    0    0    0    0    0    0    0    0    0    0
     0    0    1    0    0    0    0 2164    0    0    0    0    0]
 [   0   26    2  103    0    5    1    1    2    0    0   10    2   13
     6    2   15   17    2    0    2    1 1956    0    1    1    0]
 [   0    0    0    4    0    0    0    0    0    0    0    1    4    0
     0    0    1    1    1    0    1    0    1 2154    0    0    0]
 [   0    6    0   16    0    0    1    0    1    0    0    9    3    1
     4    1    8   37   11    0    0    0   12    0 2057    1    0]
 [   0    0    0    1    0   11    0    0    7    0    0    0    0    0
     0    0    0    2    0    0    1    0    0    0    0 2146    0]
 [   0    6    0    3    0    1    0    0    0    0    0    2    1    1
     0    1    0    1    0    0    0    0    1    1    2    0 2148]]
```

REPORT

- With the cross-validation scores ranging from 0.666 to 0.679, illustrates that the different categorical variables achieved varying levels of accuracy in classifying the dataset. 0.67 mean score shows the model has a fair degree of accuracy across the categorical variables, while the 0.68 train score indicates that the model fits the training data reasonably well.
- According to the classification report, the model performed well across most classes, as precision range from 0.6 to 1, recall 0.7 to 1, and F1-score that considers both false positives and false negatives are from 0.7 to 1. The model's accuracy of 0.95 indicates that it properly classified 95% of the occurrences.
- By analyzing the diagonal values for each class in the confusion matrix and bearing in mind that the higher the true positives, the better the variable is at correctly classifying instances, Platform variable performed well in classifying the dataset. For example, it correctly classified 2168 instances in the first class, accurately classified 1898 instances in the second class, 2139 instances in the third class, 1754 instances in the fourth class etc.

Genre as Target

In [106]:

```
# create an instance of OneHotEncoder
onehot_encoder = OneHotEncoder(sparse=False)

# Applying One-hot encoding on 'Rating' and 'Platform' column
rating_platform_encoded = onehot_encoder.fit_transform(video_games_model[['Rating',

# Getting feature names for 'Genre' column
features = onehot_encoder.get_feature_names_out(['Rating', 'Platform'])

# Creating a DataFrame with the one-hot encoded 'Rating' and 'Platform' columns
rating_platform_onehot_encoded_df = pd.DataFrame(data= rating_platform_encoded, colu

# Creating an instance of Labelencoder
label_encoder = LabelEncoder()

# Converting the target variable to numerical using Lable encoding.
genre_labeled = label_encoder.fit_transform(video_games_model['Genre'])

# Creating a DataFrame with the numerical 'Genre' column
genre_labeled_df = pd.DataFrame(genre_labeled, columns=['Genre'])

# Combining the one-hot encoded 'Rating' and numerical 'Genre' DataFrames with 'obje
data_class_gen = pd.concat([rating_platform_onehot_encoded_df, genre_labeled_df, obj
data_class_gen
```

Out[106]:

	Rating_AO	Rating_E	Rating_E10+	Rating_EC	Rating_K-A	Rating_M	Rating_RP	Rati
0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
3	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
...	...	...	...	...	...	...	...	
16714	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
16715	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
16716	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
16717	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
16718	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

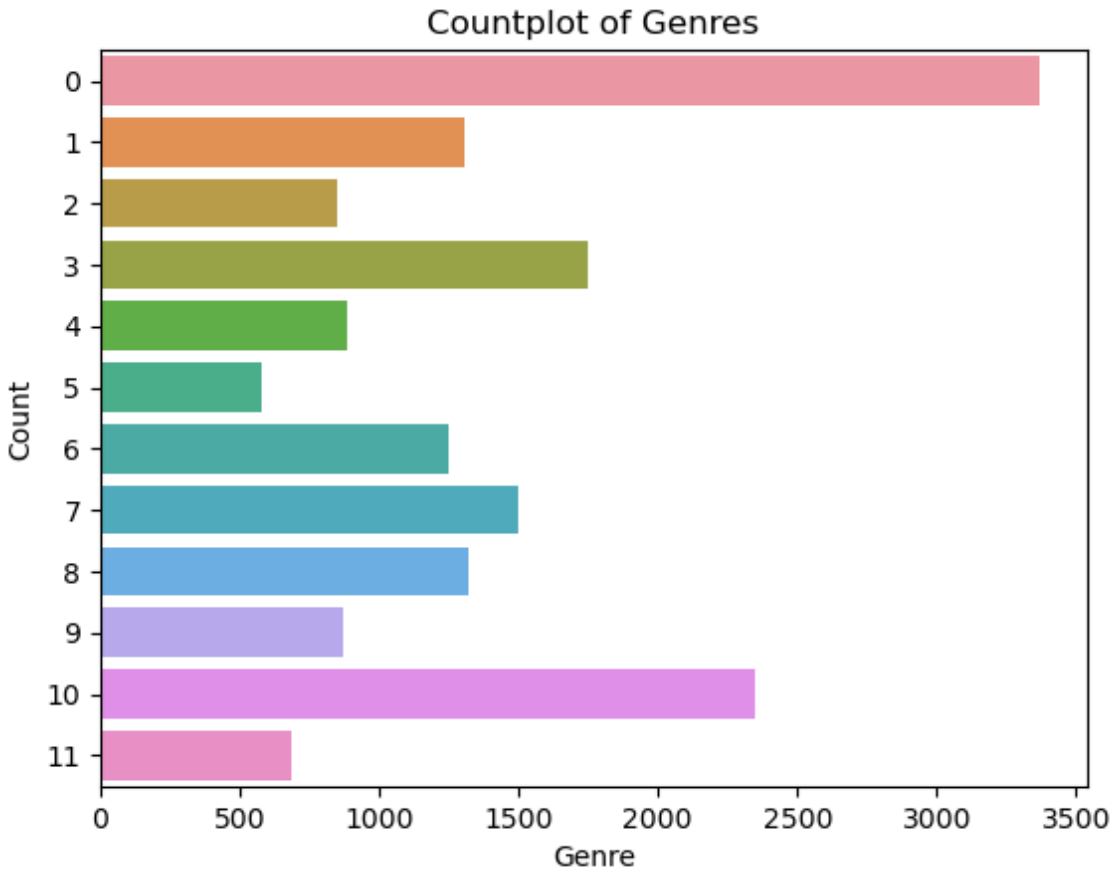
16719 rows × 46 columns

In [107]:

```
# Split data into input(X) and output(Y) variables
X = data_class_gen.drop(['Genre'],axis=1)
y1= data_class_gen['Genre']
```

In [108]:

```
#Plotting of the target(Genre) variable.
sns.countplot(y='Genre', data=data_class_gen)

plt.title('Countplot of Genres')
plt.xlabel('Genre')
plt.ylabel('Count')

#Show the plot
plt.show()
```



- The plot is not equally distributed

APPLYING SMOTE

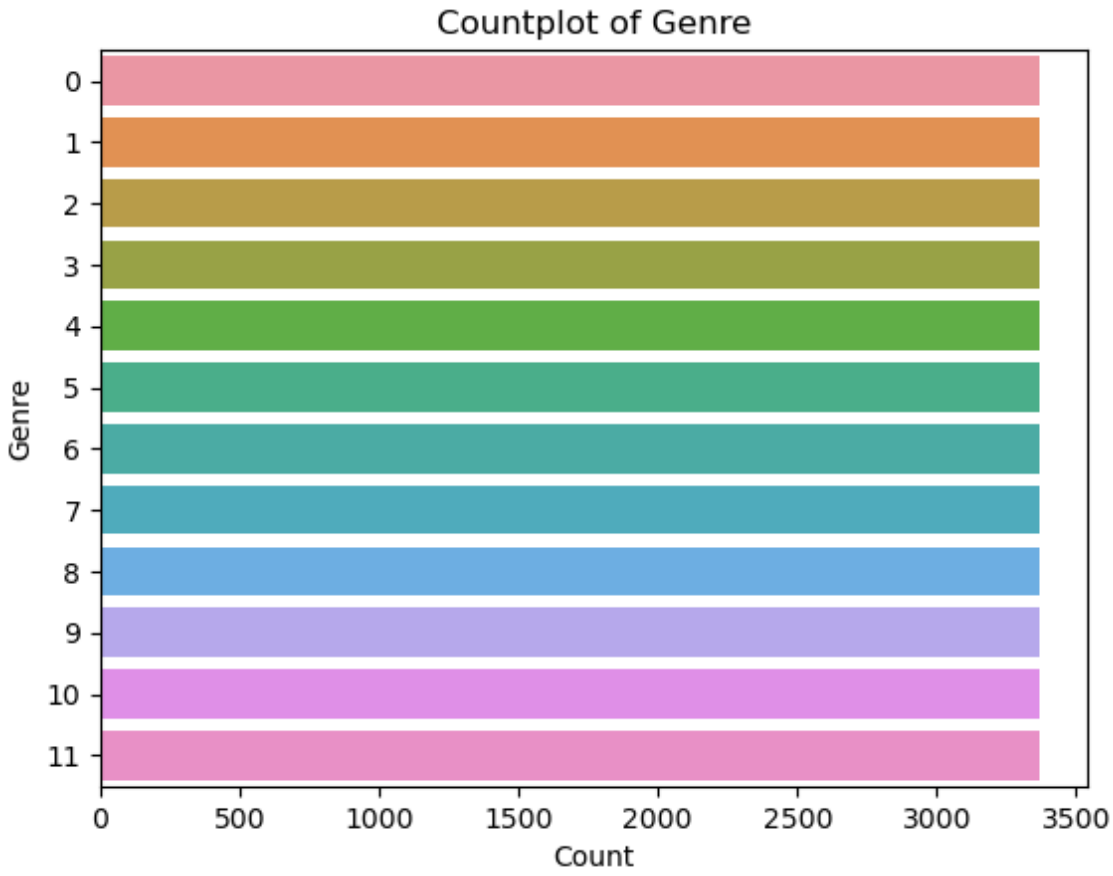
- Since our target class label data is not equally distributed, Synthetic Minority Oversampling Technique (SMOTE) will assist us to balance it.

In [109]:

```
# import SMOTE
from imblearn.over_sampling import SMOTE
sm = SMOTE(random_state=42, k_neighbors = 3) # The object is created
# apply SMOTE to resample the dataset
X_res, y_res = sm.fit_resample(X, y1) # The object is applied
# reassigning the balanced dataset to X,y
X, y = X_res, y_res

# Plot of the dataset
ax = sns.countplot(y=y, data = data_class_gen )
# set the labels for the axes and the title
ax.set_xlabel('Count')
ax.set_ylabel('Genre')
ax.set_title('Countplot of Genre')
```

Out[109]:

```
Text(0.5, 1.0, 'Countplot of Genre')
```



- Now it has been distributed equally

In [110]:

```
clf = DecisionTreeClassifier() #Initializing the classifier

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

#Perform cross_validation

scores = cross_val_score(clf,X,y,cv=cv)

#Fit classifier on the dataset.

clf.fit(X, y)

#Making predictions on the test data.

y_pred1 = clf.predict(X)
```

In [111]:

```
y_pred1
```

Out[111]:

```
array([10,  4,  6, ...,  1, 11, 11])
```

In [112]:

```
# Prints the mean and standard deviation of scores
print('Cross-validation scores:', scores)
print('Mean score:', scores.mean())
print('Standard deviation:', scores.std())
print('Train score:', scores.mean() + 2*scores.std())
```

```
Cross-validation scores: [0.48301001 0.48239219 0.49548993 0.48226863 0.48
591201]
Mean score: 0.4858145536145277
Standard deviation: 0.0050154567460838325
Train score: 0.49584546710669536
```

In [113]:

```
# Prints the Classification report
print('Classification Report:')
print(classification_report(y, y_pred1))
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.86      0.87      3372
           1       0.65      0.93      0.77      3372
           2       0.91      0.92      0.92      3372
           3       0.87      0.81      0.84      3372
           4       0.97      0.96      0.97      3372
           5       0.95      0.95      0.95      3372
           6       0.97      0.94      0.95      3372
           7       0.92      0.83      0.87      3372
           8       0.96      0.95      0.95      3372
           9       0.95      0.88      0.91      3372
          10       0.95      0.86      0.90      3372
          11       0.95      0.90      0.93      3372

    accuracy                           0.90     40464
   macro avg       0.91      0.90      0.90     40464
weighted avg       0.91      0.90      0.90     40464
```

In [114]:

```
# Prints the confusion Matrix
print('Confusion Matrix:')
print(confusion_matrix(y, y_pred1))
```

```
Confusion Matrix:
[[2895  242   29   33   12   14   12   50   18   14   26   27]
 [  37 3136   42   41    2   10    7   43   16   10   14   14]
 [  15  159 3113   14    7    1    7   15   23    1   10    7]
 [  71  340   37 2744   13   36    8   15    9   57   28   14]
 [  22   29   10   25 3252    2    6    6    3    3   12    2]
 [  29   33    8   36    4 3207    6    9    8   20    3    9]
 [  33   39   19   31   20   13 3164    5   17    2   16   13]
 [  54  316   58   44    6   18   11 2813   12   12   21    7]
 [  31   48   20   16    4    5    6   13 3204    2    6   17]
 [  19  194   22   71    3   25    6   28    8 2956   13   27]
 [  70  106   38   87   20   22   36   46   18   33 2884   12]
 [  32  170   13   24    3   31    4   31    7    5   16 3036]]
```

REPORT

- From the classification report, the model performed relatively well across most classes, with all classes having the same number of instances (support) of 3372 in the dataset, as precision range from 0.65 to 0.97, recall 0.81 to 0.96, and F1-score that is the harmonic average of precision and recall is from 0.77 to 0.97. The model's accuracy, macro average and weighted average F1-scores are all 0.90, indicating a similar overall performance as well as 90% correct classification of the instances.
- By examining the confusion matrix, we can see the patterns of misclassifications made by the model. The diagonal elements represents the correctly classified instances for each class, while the off-diagonal elements represent misclassifications.Genre variable did not perform well in classifying the

dataset. For example, it misclassified 242 instances in the first class, 159 instances in the third class, 340 instances in the third class etc.

Rating as Target

In [115]:

```
video_games_model['Rating'].value_counts()
```

Out[115]:

```
Unknown    6769
E          3991
T          2961
M          1563
E10+       1420
EC            8
K-A           3
RP            3
AO            1
Name: Rating, dtype: int64
```

In [116]:

```
'''
As there are values less than 4, k-neighbors will not smote the column. Hence
I will replace the values less than 3 with the mode of the column
'''
video_games_model['Rating'] = video_games_model['Rating'].replace(['AO','K-A', 'RP']
```

In [117]:

```
# Confirming the changes
video_games_model['Rating'].value_counts()
```

Out[117]:

```
Unknown    6776
E          3991
T          2961
M          1563
E10+       1420
EC            8
Name: Rating, dtype: int64
```

In [118]:

```
# create an instance of OneHotEncoder
onehot_encoder = OneHotEncoder(sparse=False)

# Applying One-hot encoding on 'Genre' and 'Platform' column
genre_platform_encoded = onehot_encoder.fit_transform(video_games_model[['Genre', 'P

# Getting feature names for 'Rating' column
features = onehot_encoder.get_feature_names_out(['Genre', 'Platform'])

# Creating a DataFrame with the one-hot encoded 'Rating' and 'Platform' columns
genre_platform_onehot_encoded_df = pd.DataFrame(data= genre_platform_encoded, columr

# Creating an instance of Labelencoder
label_encoder = LabelEncoder()

# Converting the target variable to numerical using Lable encoding.
rating_labeled = label_encoder.fit_transform(video_games_model['Rating'])

# Creating a DataFrame with the numerical 'Rating' column
rating_labeled_df = pd.DataFrame(rating_labeled, columns= ['Rating'])

# Combining the one-hot encoded '|Genre'and 'Platform' and numerical 'Rating' DataFr
data_class_rating = pd.concat([genre_platform_onehot_encoded_df, rating_labeled_df,
data_class_rating
```

Out[118]:

	Genre_Action	Genre_Adventure	Genre_Fighting	Genre_Misc	Genre_Platform	Genre_
0	0.0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	1.0	
2	0.0	0.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.0	
...	...	...	...	...	...	
16714	1.0	0.0	0.0	0.0	0.0	
16715	0.0	0.0	0.0	0.0	0.0	
16716	0.0	1.0	0.0	0.0	0.0	
16717	0.0	0.0	0.0	0.0	1.0	
16718	0.0	0.0	0.0	0.0	0.0	

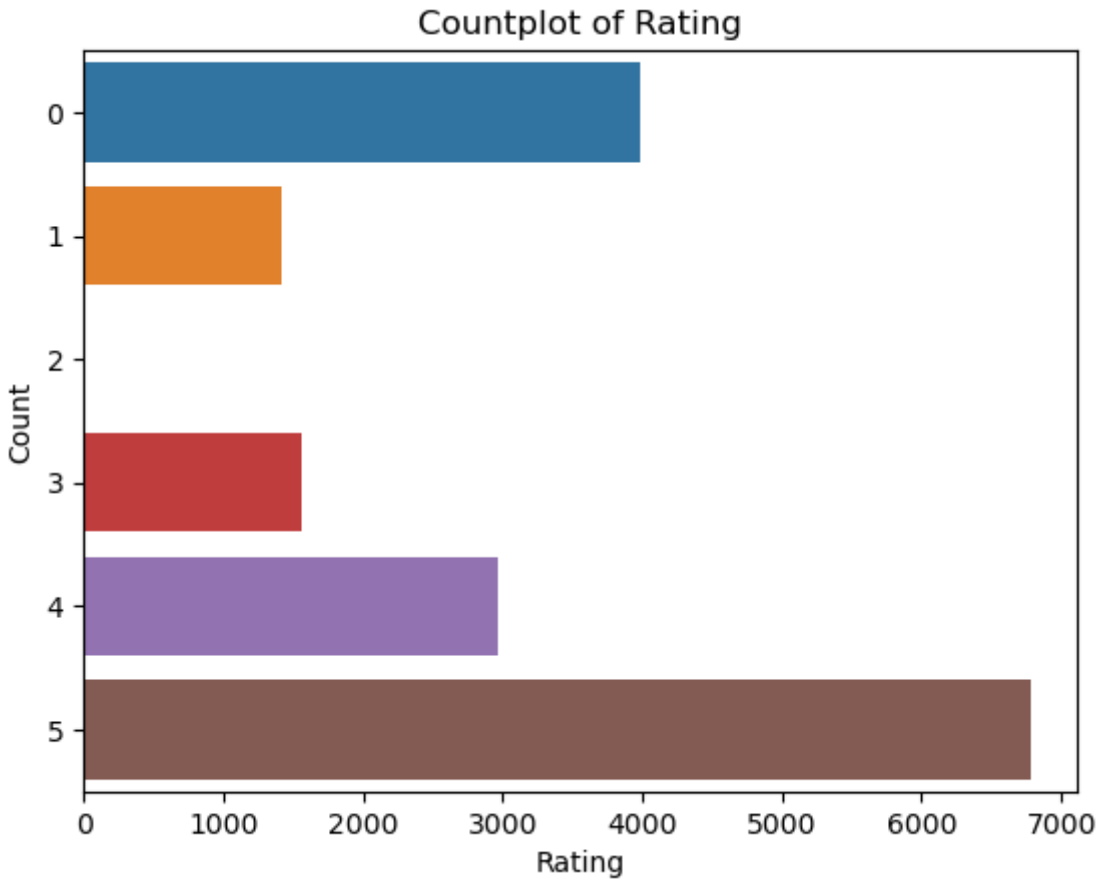
16719 rows × 49 columns

In [119]:

```
# Split data into input(X) and output(Y) variables
X = data_class_rating.drop(['Rating'],axis=1)
y2= data_class_rating['Rating']
```

In [120]:

```
#Plotting of the target(Genre) variable.
sns.countplot(y='Rating', data=data_class_rating)

plt.title('Countplot of Rating')
plt.xlabel('Rating')
plt.ylabel('Count')

#Show the plot
plt.show()
```



- The plot is not equally distributed

APPLYING SMOTE

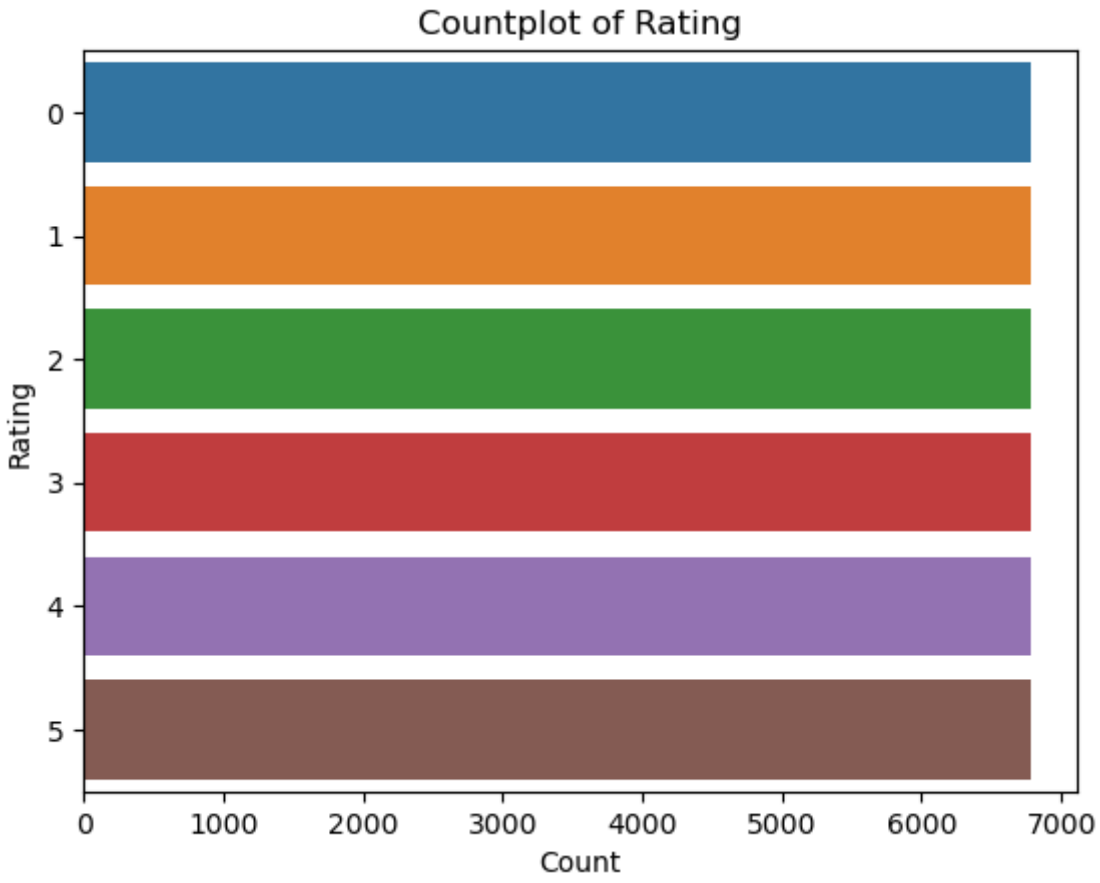
- Since our target class label data is not equally distributed, Synthetic Minority Oversampling Technique (SMOTE) will assist us to balance it.

In [121]:

```
# SMOTE imported
sm = SMOTE(random_state=42, k_neighbors = 3) # The object is created
# apply SMOTE to resample the dataset
X_res, y_res = sm.fit_resample(X, y2) # The object is applied
# reassigning the balanced dataset to X,y
X, y = X_res, y_res

# Plot of the dataset
ax = sns.countplot(y=y, data = data_class_rating )
# set the labels for the axes and the title
ax.set_xlabel('Count')
ax.set_ylabel('Rating')
ax.set_title('Countplot of Rating')
```

Out[121]:

```
Text(0.5, 1.0, 'Countplot of Rating')
```



- Now it has been distributed equally

In [122]:

```
clf = DecisionTreeClassifier() #Initializing the classifier

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

#Perform cross_validation

scores = cross_val_score(clf,X,y,cv=cv)

#Fit classifier on the dataset.

clf.fit(X, y)

#Making predictions on the test data.

y_pred2 = clf.predict(X)
```

In [123]:

```
y_pred2
```

Out[123]:

```
array([0, 5, 0, ..., 4, 4, 4])
```

In [124]:

```
# Prints the mean and standard deviation of scores
print('Cross-validation scores:', scores)
print('Mean score:', scores.mean())
print('Standard deviation:', scores.std())
print('Train score:', scores.mean() + 2*scores.std())
```

```
Cross-validation scores: [0.80632071 0.8000246  0.79781085 0.787603   0.80
519001]
Mean score: 0.7993898334593946
Standard deviation: 0.006685875557654409
Train score: 0.8127615845747035
```

In [125]:

```
# Prints the Classification report
print('Classification Report:')
print(classification_report(y, y_pred2))
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.99      0.98      6776
           1       1.00      0.99      0.99      6776
           2       1.00      1.00      1.00      6776
           3       1.00      1.00      1.00      6776
           4       1.00      0.99      1.00      6776
           5       0.99      0.97      0.98      6776

    accuracy                           0.99     40656
   macro avg       0.99      0.99      0.99     40656
weighted avg       0.99      0.99      0.99     40656
```

In [126]:

```
# Prints the confusion Matrix
print('Confusion Matrix:')
print(confusion_matrix(y, y_pred2))
```

```
Confusion Matrix:
[[6734    4    0    0    0   38]
 [  31 6736    0    0    0    9]
 [   0    0 6776    0    0    0]
 [   0    0    0 6775    0    1]
 [  21    7    0    0 6735   13]
 [ 139   21    0    2   14 6600]]
```

REPORT

- **Based on this classification report, Rating performed best in classifying the dataset.** The model demonstrates excellent performance with high precision of 0.97 to 1.00, high recall of 0.97 to 1.00, and high F1-scores of 0.98 to 1.00 across all classes. It achieves an overall accuracy of 0.99, suggesting that it effectively classifies the instances in the dataset.
- By analyzing the diagonal values for each class in the confusion matrix and bearing in mind that the higher the true positives, the better the variable is at correctly classifying instances, **Rating variable performed best in classifying the dataset**. For example, it correctly classified 6734 instances in the first class, accurately classified 6736 instances in the second class, 6776 instances in the third class, 6775 instances in the fourth class etc.

Question 2e.

- **How did you check whether your models did not overfit?**

In [131]:

```
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat

# Fitting a logistic regression model on the training data
log_model = LogisticRegression()
log_model.fit(X_train, y_train)

# Evaluating the model on the training set
train_pred = log_model.predict(X_train)
train_acc = accuracy_score(y_train, train_pred)

# Evaluate the model on the testing set
test_preds = log_model.predict(X_test)
test_acc = accuracy_score(y_test, test_preds)

```

In [132]:

```
# Printing the training and testing accuracies
print('The training accuracy is :', train_acc)
print('The testing accuracy is:', test_acc)
```

```
The training accuracy is : 0.5293629319886852
The testing accuracy is: 0.5241023118544024
```

REPORT

The training accuracy of 0.5293629319886852 indicates that my models successfully predicted the target variable 52.94% instances on the training, while the testing accuracy of 0.5241023118544024 shows that my models correctly predicted the target variable 52.41% instances on the testing.

As the training accuracy is not significantly higher than the testing accuracy, this shows that my models did not overfit.

The similar accuracy of both the training and testing which is between 52-53%, indicates that my models performance is not significantly different on the training and testing data, hence, my models can be said to have had a good generalization performance.

In addition, I specifically used Decision Tree Regressor as one of my regressors to evaluate my models' performance despite being aware of how prone it is to overfitting. However, this regressor did not overfit and was also the overall best regressor.

Question 2f.

- **Can your classification models be deployed in practice based on their performances? Explain.**

- Yes, my classification model can be deployed in practice as it has demonstrated high accuracy of 90% and above across different variables, high precision, high recall, high F1-score, and confusion matrix

which showed promising results in the low number of misclassifications suggesting that the model can

Question 2g.

- **In the video game dataset, use a relevant categorical variable and other relevant non-categorical variables to form groups at each instance. By employing internal and external evaluation metrics, determine which categorical variable best describes the groups formed.**

Using Rating as the relevant categorical variable

K-Means Clustering Algorithm

In [133]:

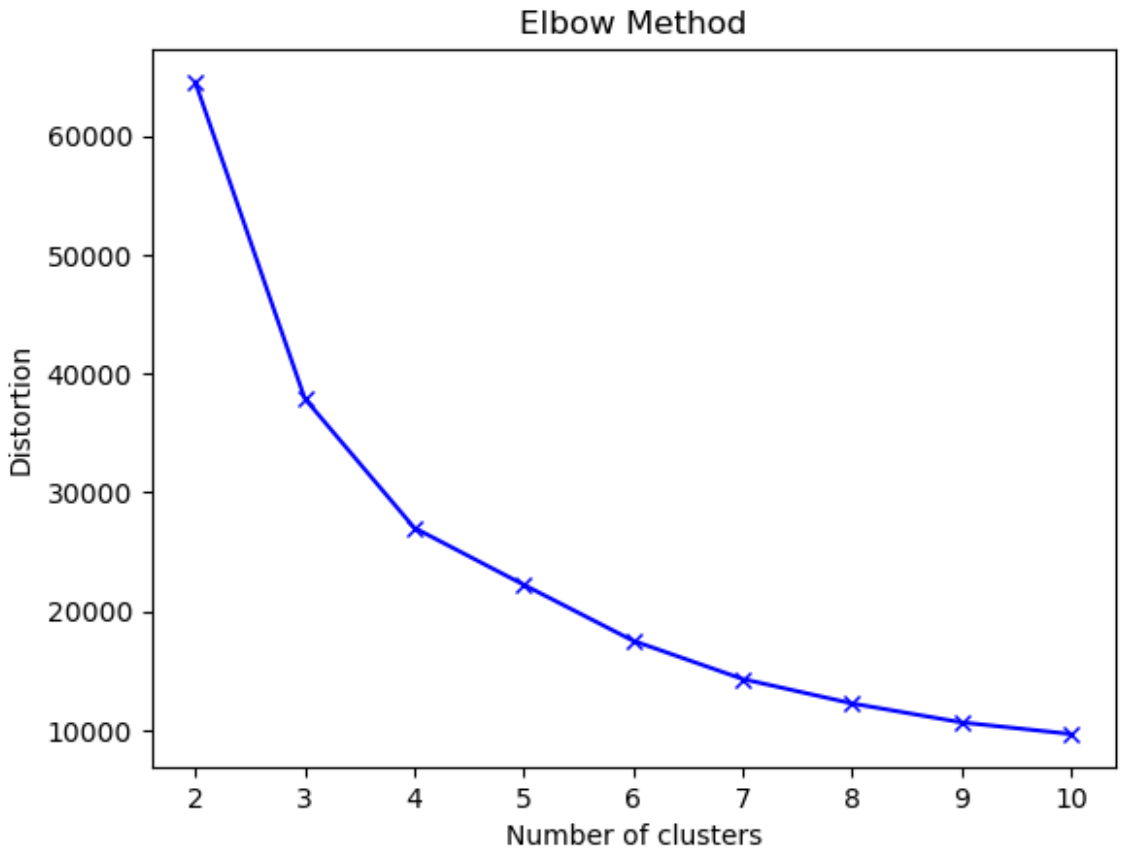
```
# import kmeans library
from sklearn.cluster import KMeans
```

In [134]:

```
# Combine categorical and non-categorical variables
cat_num_df = pd.concat([object_float_df, cat_data], axis=1)

# Select relevant columns for clustering
clustering_data = data_class_rating[['Rating', 'NA_Sales', 'JP_Sales', 'Other_Sales'

# Scale the numerical variables
scale = StandardScaler()
scaled_cluster = scale.fit_transform(clustering_data)


```

In [135]:

```
# Perform elbow method to determine the number of clusters
distortion = []
K = range(2, 11)
for k in K:
    kmeans = KMeans(n_clusters=k, random_state = 42)
    kmeans.fit(clustering_data)
    distortion.append(kmeans.inertia_)

# Plot the elbow curve
plt.plot(K, distortion, 'bx-')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.title('Elbow Method')
plt.savefig('Elbow Method.png')
plt.show()

# fitting the k-means algorithm
kmeans = KMeans(n_clusters=3, random_state=42)
C_labels = kmeans.fit_predict(scaled_cluster)

# Perform K-means clustering with chosen number of clusters
kmeans_cluster = KMeans(n_clusters=3, random_state=42)
C_labels = kmeans_cluster.fit_predict(scaled_cluster)

C_labels
```



Out[135]:

```
array([1, 1, 1, ..., 0, 0, 0])
```

In [136]:

```
# Get the required class labels
y_true = cat_num_df['Rating']
```

In [137]:

```
print('\n')
print('External Evaluation Measures')
print('*******************************')

# Calculate the v-measure score
v_measure = v_measure_score(y_true, C_labels)
print('V-measure Score:', round(v_measure,2))

# Calculate the Rand index score
ri_score = adjusted_rand_score(y_true, C_labels)
print('Rand Index Score:', round(ri_score, 2))

# Calculate the mutual information score
ami_score = adjusted_mutual_info_score(y_true, C_labels)
print('Mutual Information Score:', round(ami_score,2))

print('\n')
print('Internal Evaluation Measures')
print('*******************************')

# Calculate the Davies-Bouldin index
db_score = davies_bouldin_score(scaled_cluster, C_labels)
print('Davies-Bouldin Index:', round(db_score,2))

# Calculate the Silhouette coefficient
s_score = silhouette_score(scaled_cluster, C_labels)
print('Silhouette Coefficient:', round(s_score,2))

# Calculate the Calinski Harabasz Score
c_h_score = calinski_harabasz_score(scaled_cluster, C_labels)
print('Calinski Harabasz Score:', round(c_h_score,2))
```

```
External Evaluation Measures
*******************************
V-measure Score: 0.56
Rand Index Score: 0.45
Mutual Information Score: 0.56


Internal Evaluation Measures
*******************************
Davies-Bouldin Index: 0.82
Silhouette Coefficient: 0.55
Calinski Harabasz Score: 7090.24
```

REPORT

- Based on the external evaluation metrics, a higher score suggests that the groups and the external criteria are more aligned. The V-measure and Mutual Information scores in this scenario are both 0.56, indicating a moderate amount of agreement between the created groups and the external criteria. The Rand Index score of 0.45 implies that there is less agreement.
- Based on the internal measures, lower Davies-Bouldin Index and Silhouette Coefficient values suggest better group cohesion and separation, while higher Calinski Harabasz Score values imply better-defined and more compact groups. The Davies-Bouldin Index of 0.82 indicates moderate cohesiveness and separation in this example, the Silhouette Coefficient of 0.55 indicates a medium degree of differentiation between the groups, and the Calinski Harabasz Score of 7090.24 indicates well-defined and compact groups.

With the above evaluation metrics, I conclude that using K-Means Clustering the formed groups of Rating and the selected non-categorical variables show moderate to fair agreement with the external criteria and exhibit moderate to good internal quality.

DBSCAN Clustering Algorithm

In [138]:

```
# Import the required library
from sklearn.cluster import DBSCAN
```

In [139]:

```
#Instantiating the model
dbscan = DBSCAN(eps=0.2,min_samples=5)

#Fit the model and predict the label
D_labels = dbscan.fit_predict(clustering_data)

```

In [140]:

```
# Calculating the Evaluation Metrics

print('\n')
print('External Evaluation Measures')
print('**************************************')

# Calculate V measure score
v_measure = v_measure_score(y_true, D_labels)
print('V-measure score:', round(v_measure,2))

# Calculate Rand Index score
ri_score = adjusted_rand_score(y_true, D_labels)
print('Rand Index Score:', round(ri_score, 2))

# Calculate the Mutual information score
ami_score = adjusted_mutual_info_score(y_true, D_labels)
print('Mutual information score:', round(ami_score,2))

print('\n')
print('Internal Evaluation Measures')
print('**************************************')

# Calculate the Davies-Bouldin index
db_score = davies_bouldin_score(clustering_data, D_labels)
print('Davies Bouldin Index:', round(db_score,2))

# Calculate the Silhouette coefficient
s_score = silhouette_score(clustering_data, D_labels)
print('Silhouette Score:', round(s_score, 2))

# Calculate the Calinski Harabasz Score
c_h_score = calinski_harabasz_score(clustering_data, D_labels)
print('Calinski Harabasz Score:', round(c_h_score, 2))
```

```
External Evaluation Measures
**************************************
V-measure score: 0.89
Rand Index Score: 0.92
Mutual information score: 0.89


Internal Evaluation Measures
**************************************
Davies Bouldin Index: 1.39
Silhouette Score: 0.52
Calinski Harabasz Score: 1683.47
```

REPORT

- The external evaluation metrics indicate a high level of agreement between the formed groups and the external criteria, suggesting that the variables effectively describe the groups.
- The internal evaluation metrics reveal moderate to good cohesion, separation, and compactness of the groups. While the Davies Bouldin Index of 1.39 suggests moderate cohesion and separation, the

Silhouette Score of 0.52 indicates a moderate degree of distinction between the groups, and the Calinski Harabasz Score of 1683.47 indicates well-defined and compact groups.

In summary, using DBSCAN and Rating as the categorical variable, the groups demonstrate high level of

Using Genre as the relevant categorical variable

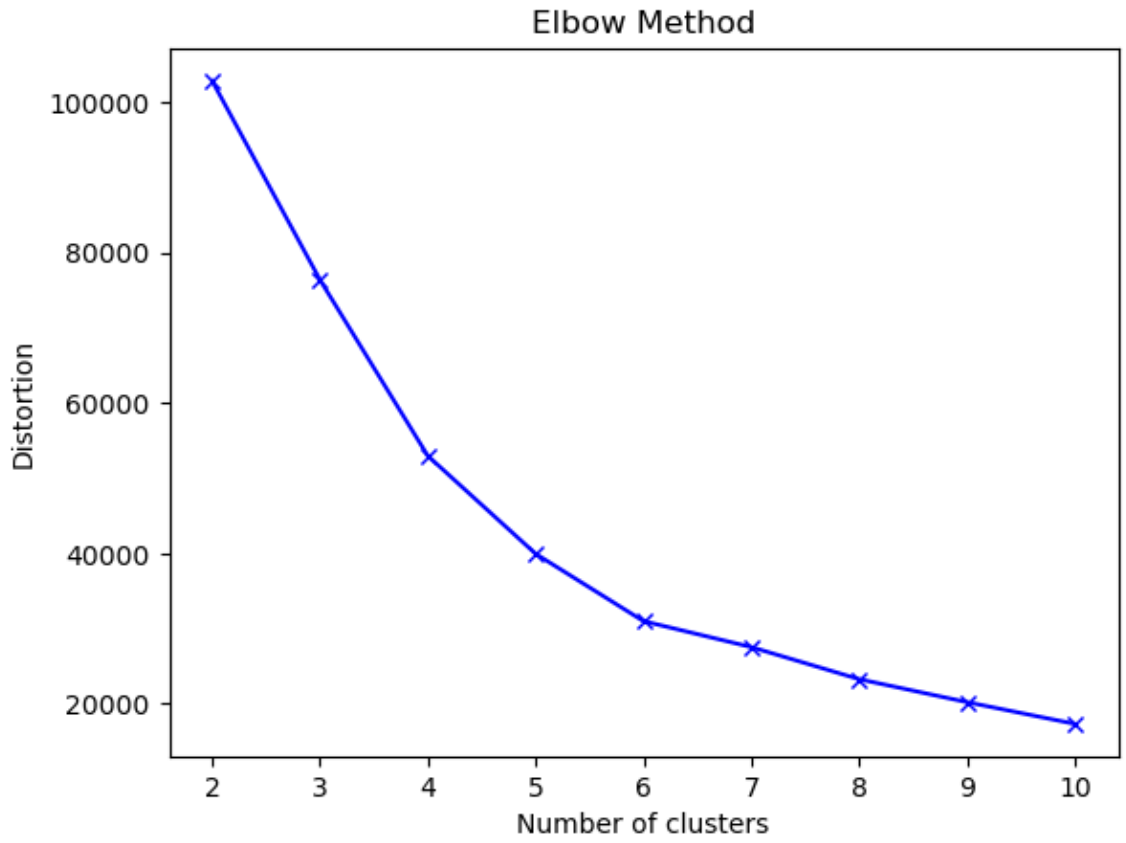
K-Means Clustering Algorithm

In [141]:

```
# Select relevant columns for clustering
clustering_data_genre = data_class_gen[['Genre', 'NA_Sales', 'JP_Sales', 'Other_Sale

# Scale the numerical variables
scale = StandardScaler()
scaled_cluster_genre = scale.fit_transform(clustering_data_genre)
```

In [142]:

```
# Perform elbow method to determine the number of clusters
distortion = []
K = range(2, 11)
for k in K:
    kmeans = KMeans(n_clusters=k, random_state = 42)
    kmeans.fit(clustering_data_genre)
    distortion.append(kmeans.inertia_)

# Plot the elbow curve
plt.plot(K, distortion, 'bx-')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.title('Elbow Method')
plt.savefig('Elbow Method_genre.png')
plt.show()

# fitting the k-means algorithm
kmeans = KMeans(n_clusters=3, random_state=42)
C_labels = kmeans.fit_predict(scaled_cluster_genre)

# Perform K-means clustering with chosen number of clusters
kmeans_cluster = KMeans(n_clusters=3, random_state=42)
C_labels = kmeans_cluster.fit_predict(scaled_cluster_genre)

C_labels
```



Out[142]:

```
array([1, 1, 1, ..., 0, 0, 0])
```

In [143]:

```
# Get the required class labels
y_true1 = cat_num_df['Genre']
```

In [144]:

```
print('\n')
print('External Evaluation Measures')
print('*******************************')

# Calculate the v-measure score
v_measure = v_measure_score(y_true1, C_labels)
print('V-measure Score:', round(v_measure,2))

# Calculate the Rand index score
ri_score = adjusted_rand_score(y_true1, C_labels)
print('Rand Index Score:', round(ri_score, 2))

# Calculate the Mutual Information score
ami_score = adjusted_mutual_info_score(y_true1, C_labels)
print('Mutual Information Score:', round(ami_score,2))

print('\n')
print('Internal Evaluation Measures')
print('*******************************')

# Calculate the Davies-Bouldin index
db_score = davies_bouldin_score(scaled_cluster_genre, C_labels)
print('Davies-Bouldin Index:', round(db_score,2))

# Calculate the Silhouette coefficient
s_score = silhouette_score(scaled_cluster_genre, C_labels)
print('Silhouette Coefficient:', round(s_score,2))

# Calculate the Calinski Harabasz Score
c_h_score = calinski_harabasz_score(scaled_cluster_genre, C_labels)
print('Calinski Harabasz Score:', round(c_h_score,2))
```

```
External Evaluation Measures
*******************************
V-measure Score: 0.0
Rand Index Score: 0.0
Mutual Information Score: 0.0


Internal Evaluation Measures
*******************************
Davies-Bouldin Index: 0.95
Silhouette Coefficient: 0.74
Calinski Harabasz Score: 6654.58
```

REPORT

- The external evaluation metrics of 0.0 indicate a lack of agreement between the formed groups and the external criteria, suggesting that the variables do not effectively describe the groups.

- The internal evaluation measures show that the groupings have rather excellent cohesion, isolation, and compactness. The Davies Bouldin Index of 0.95 indicates adequate separation and compactness, whilst the Silhouette Coefficient of 0.74 indicates a relatively well-defined differentiation between the groupings. The Calinski Harabasz Score of 6654.58 suggests that the groups are relatively compact.

In summary, using KMeans with Genre as the categorical variable, the formed groups do not exhibit agreement with the external criteria. However, they demonstrate reasonably good internal quality in terms of cohesion, separation, and compactness.

DBSCAN Clustering Algorithm

In [145]:

```
#Instantiating the model
dbscan = DBSCAN(eps=0.2,min_samples=5)

#Fit the model and predict the label
D_labels = dbscan.fit_predict(clustering_data_genre)

```

In [146]:

```
# Calculating the Evaluation Metrics

print('\n')
print('External Evaluation Measures')
print('**************************************')

# Calculate V measure score
v_measure = v_measure_score(y_true1, D_labels)
print('V-measure score:', round(v_measure,2))

# Calculate Rand Index score
ri_score = adjusted_rand_score(y_true1, D_labels)
print('Rand Index Score:', round(ri_score, 2))

# Calculate the Mutual information score
ami_score = adjusted_mutual_info_score(y_true1, D_labels)
print('Mutual information score:', round(ami_score,2))

print('\n')
print('Internal Evaluation Measures')
print('**************************************')

# Calculate the Davies-Bouldin index
db_score = davies_bouldin_score(clustering_data_genre, D_labels)
print('Davies Bouldin Index:', round(db_score,2))

# Calculate the Silhouette coefficient
s_score = silhouette_score(clustering_data_genre, D_labels)
print('Silhouette Score:', round(s_score, 2))

# Calculate the Calinski Harabasz Score
c_h_score = calinski_harabasz_score(clustering_data_genre, D_labels)
print('Calinski Harabasz Score:', round(c_h_score, 2))
```

```
External Evaluation Measures
**************************************
V-measure score: 0.9
Rand Index Score: 0.89
Mutual information score: 0.9


Internal Evaluation Measures
**************************************
Davies Bouldin Index: 1.24
Silhouette Score: 0.52
Calinski Harabasz Score: 3180.04
```

REPORT

- The high values of these external evaluation metrics indicate a strong agreement between the formed groups and the external criteria, suggesting that the selected categorical and non-categorical variables effectively describe the groups.
- The Davies Bouldin Index of 1.24 suggests a moderate level of separation and compactness of the groups, with a lower value indicating better results. The Silhouette Score of 0.52 indicates a moderate

degree of distinction between the clusters, with values closer to 1 indicating better-defined clusters. The Calinski Harabasz Score of 3180.04 reflects the compactness of the clusters, with higher scores indicating more compact and well-separated groups.

In summary, using DBSCAN and Genre as the categorical variable, the formed groups have strong agreement with external criteria, demonstrating that these variables accurately define the groups. The internal evaluation metrics indicate moderate levels of separation, compactness, and distinctiveness among the groups, indicating reasonable quality in terms of the generated clusters.

From my analysis above, Genre is the categorical variable that best describes the formed groups.

Using Platform as the relevant categorical variable

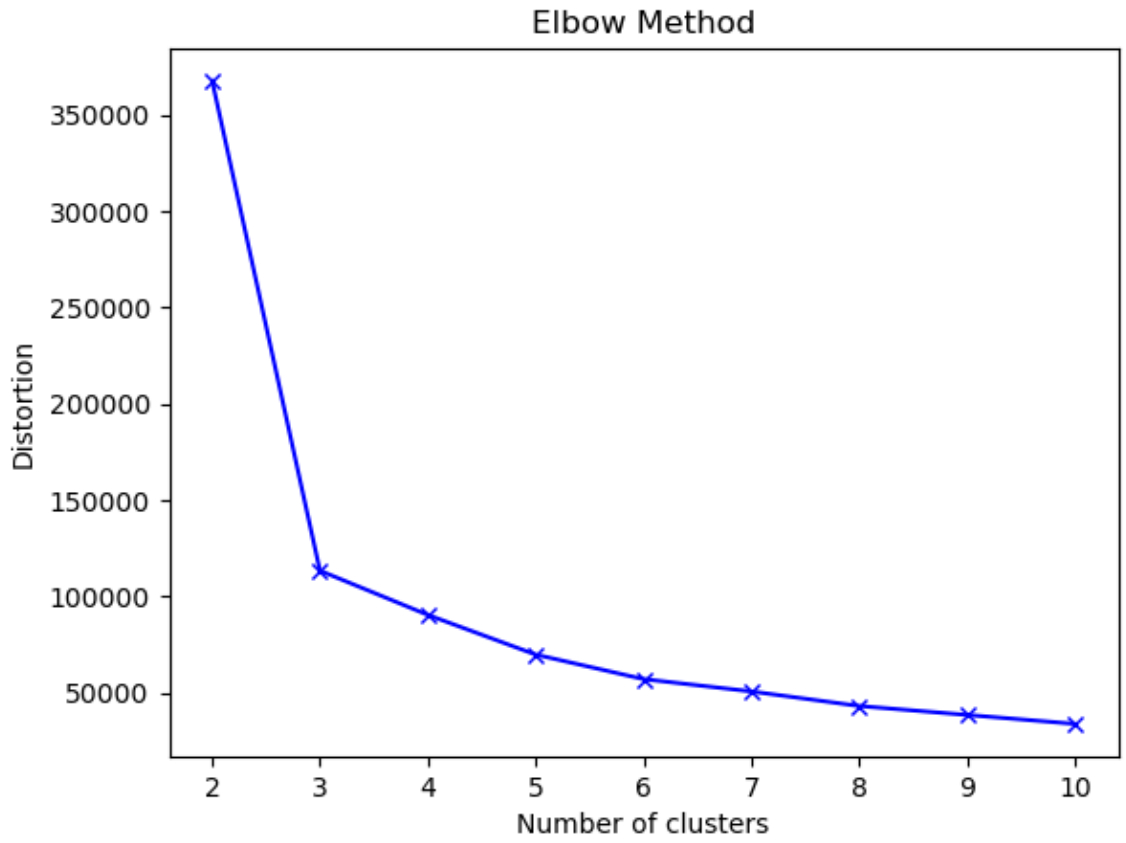
K-Means Clustering Algorithm

In [147]:

```
# Select relevant columns for clustering
clustering_data_platform = data_class_plat[['Platform', 'NA_Sales', 'JP_Sales', 'Oth

# Scale the numerical variables
scale = StandardScaler()
scaled_cluster_platform = scale.fit_transform(clustering_data_platform)
```

In [148]:

```
# Perform elbow method to determine the number of clusters
distortion = []
K = range(2, 11)
for k in K:
    kmeans = KMeans(n_clusters=k, random_state = 42)
    kmeans.fit(clustering_data_platform)
    distortion.append(kmeans.inertia_)

# Plot the elbow curve
plt.plot(K, distortion, 'bx-')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.title('Elbow Method')
plt.savefig('Elbow Method_plat.png')
plt.show()

# fitting the k-means algorithm
kmeans = KMeans(n_clusters=3, random_state=42)
C_labels = kmeans.fit_predict(scaled_cluster_platform)

# Perform K-means clustering with chosen number of clusters
kmeans_cluster = KMeans(n_clusters=3, random_state=42)
C_labels = kmeans_cluster.fit_predict(scaled_cluster_platform)

C_labels
```



Out[148]:

```
array([1, 1, 1, ..., 0, 0, 0])
```

In [149]:

```
# Get the required class labels
y_true2 = cat_num_df['Platform']
```

In [150]:

```
print('\n')
print('External Evaluation Measures')
print('*******************************')

# Calculate the v-measure score
v_measure = v_measure_score(y_true2, C_labels)
print('V-measure Score:', round(v_measure,2))

# Calculate the Rand index score
ri_score = adjusted_rand_score(y_true2, C_labels)
print('Rand Index Score:', round(ri_score, 2))

# Calculate the Mutual Information score
ami_score = adjusted_mutual_info_score(y_true2, C_labels)
print('Mutual Information Score:', round(ami_score,2))

print('\n')
print('Internal Evaluation Measures')
print('*******************************')

# Calculate the Davies-Bouldin index
db_score = davies_bouldin_score(scaled_cluster_platform, C_labels)
print('Davies-Bouldin Index:', round(db_score,2))

# Calculate the Silhouette coefficient
s_score = silhouette_score(scaled_cluster_platform, C_labels)
print('Silhouette Coefficient:', round(s_score,2))

# Calculate the Calinski Harabasz Score
c_h_score = calinski_harabasz_score(scaled_cluster_platform, C_labels)
print('Calinski Harabasz Score:', round(c_h_score,2))
```

```
External Evaluation Measures
*******************************
V-measure Score: 0.01
Rand Index Score: 0.0
Mutual Information Score: 0.01


Internal Evaluation Measures
*******************************
Davies-Bouldin Index: 0.94
Silhouette Coefficient: 0.74
Calinski Harabasz Score: 6656.68
```

REPORT

While the internal evaluation metrics suggest reasonable quality in terms of separation, compactness, and distinctiveness among the groups, the external evaluation metrics indicate a weak agreement with external criteria. This suggests that the selected variables may not effectively describe the formed groups.

DBSCAN Clustering Algorithm

In [151]:

```
#Instantiating the model
dbscan = DBSCAN(eps=0.2,min_samples=5)

#Fit the model and predict the label
D_labels = dbscan.fit_predict(clustering_data_platform)

```

In [152]:

```
# Calculating the Evaluation Metrics

print('\n')
print('External Evaluation Measures')
print('**************************************')

# Calculate V measure score
v_measure = v_measure_score(y_true2, D_labels)
print('V-measure score:', round(v_measure,2))

# Calculate Rand Index score
ri_score = adjusted_rand_score(y_true2, D_labels)
print('Rand Index Score:', round(ri_score, 2))

# Calculate the Mutual information score
ami_score = adjusted_mutual_info_score(y_true2, D_labels)
print('Mutual information score:', round(ami_score,2))

print('\n')
print('Internal Evaluation Measures')
print('**************************************')

# Calculate the Davies-Bouldin index
db_score = davies_bouldin_score(clustering_data_platform, D_labels)
print('Davies Bouldin Index:', round(db_score,2))

# Calculate the Silhouette coefficient
s_score = silhouette_score(clustering_data_platform, D_labels)
print('Silhouette Score:', round(s_score, 2))

# Calculate the Calinski Harabasz Score
c_h_score = calinski_harabasz_score(clustering_data_platform, D_labels)
print('Calinski Harabasz Score:', round(c_h_score, 2))
```

```
External Evaluation Measures
**************************************
V-measure score: 0.91
Rand Index Score: 0.89
Mutual information score: 0.91


Internal Evaluation Measures
**************************************
Davies Bouldin Index: 1.58
Silhouette Score: 0.52
Calinski Harabasz Score: 2577.6
```

The evaluation metrics indicate that the selected category and non-categorical variables properly define the formed groups. The high scores in the external evaluation measures show a good alignment with external standards, whereas the internal evaluation measures indicate reasonable quality in terms of separation, compactness, and distinctiveness within the groupings.

In [ ]:

```
1
```

Handwritten Digits Recognition

Applying Convolutional Neural Network on the MNIST dataset

About the Dataset

Source: http://yann.lecun.com/exdb/mnist/ (http://yann.lecun.com/exdb/mnist/)

DESCRIPTION

- Number of classes: 10
- Total number of images: 70000 28x28 colour images
- Number of images per class: 7000 images per class
- Number of training images: 60000
- Number of test images: 10000

STEPS INVOLVED

1. Import necessary libraries: tensorflow, SGD, numpy, classification_report etc.

2. Data pre-processing: The dataset should contain labeled samples for both training and testing.

3. Preprocess the data: This step involves transformation of the image dataset, for example, augmentation using shift, rotation, flip methods etc.

4. Model Architecture: In Keras, a Sequential model is a linear stack of layers, and you can add layers to it using the add() method. The first layer in the model should specify the input shape.

5. Compilation stage: This step involves specifying the loss function, the optimizer, and any metrics that you want to track during training.

6.Training and Evaluation stage: Use the fit() method to train the model on the training data. You can also specify the validation data here. Use the evaluate() method to evaluate the model on the test data. This will give you the accuracy and other metrics that you specified earlier.

7. Make predictions: Use the predict() method to make predictions on new data.

8.Visualisation: Use tools like Confusion matrix, classification report, Matplotlib to visualize the results, for example, plot the training and validation loss, plot the confusion matrix, etc.

Step 1: Import the required libraries

In [1]:

```
# import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from keras.utils import np_utils
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization, Activat
from keras import regularizers
from tensorflow.keras.optimizers import SGD,Adam
from keras.optimizers import RMSprop
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from keras import callbacks
```

Step 2: Load the dataset

Load the MNIST dataset

The x_train and y_train variables contain the 60,000 training images and their corresponding labels,

while x_test and y_test contain the 10,000 test images and their labels.

mnist.load_data() function splits the MNIST dataset into 60,000 training images and 10,000 test images.

the load_data() function will return two tuples one containing the training data and the other, test data

In [2]:
```
# Load the  MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

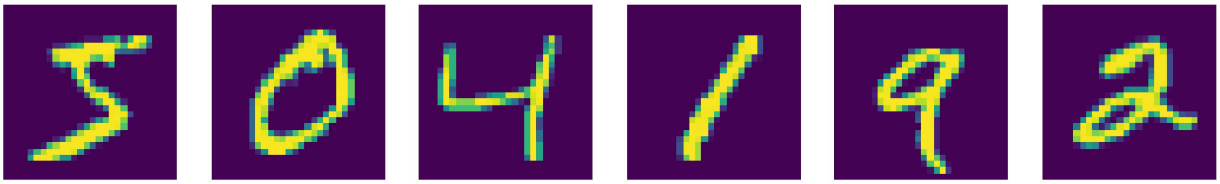
In [3]:
```
x_train.shape  # This shows the shape of one of the train images (Length, width, channel)
```

Out[3]: (60000, 28, 28)

In [4]:
```
x_test.shape # dimension of the testing data
```

Out[4]: (10000, 28, 28)

In [5]:
```
# Plot the first 6 images from the training set
fig, axs = plt.subplots(1, 6, figsize=(15, 3))
for i in range(6):
    axs[i].imshow(x_train[i])
    axs[i].axis('off')
plt.show()
```

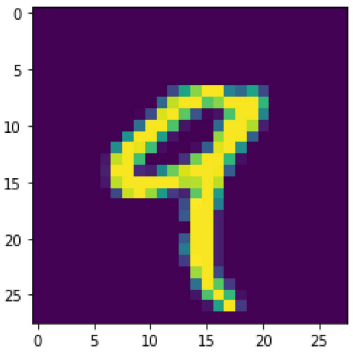


In [6]:
```
# target
np.unique(y_train)
```

Out[6]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)

In [7]:
```

plt.imshow(x_train[4])
```

Out[7]: <matplotlib.image.AxesImage at 0x16ba4bba4c0>



In [8]:
```
x_train
```

Out[8]:
```
array([[[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]],

       [[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]],

       [[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]],

       ...,

       [[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]],

       [[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]],

       [[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]]], dtype=uint8)
```

In [9]:
```
y_train
```

Out[9]:
```
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

In [10]:
```
y_train.shape
```

Out[10]:
```
(60000,)
```

Step 3: Data preprocessing

In [11]:
```
# Normalize pixel values to range 0-1
# the pixel values in an image typically range from 0 to 255
# 1 dividing all pixel values by 255 brings the pixel values to the range between 0 and 1
x_train = x_train / 255.0
x_test = x_test / 255.0
```

In [12]:
```
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
```

In [13]:

```
# Define data augmentation parameters for training set

train_datagenerator = ImageDataGenerator(
    rotation_range=20,  # Randomly rotate images up to 20 degrees
    width_shift_range=0.1,  # Randomly shift images horizontally up to 10% of the width
    height_shift_range=0.1,  # Randomly shift images vertically up to 10% of the height
    horizontal_flip=True,  # Randomly flip images horizontally
    vertical_flip=False,  # Don't randomly flip images vertically
    shear_range=0.10, # crops part of the image
    zoom_range=0.10 # #zooms the image by 10%
)
```

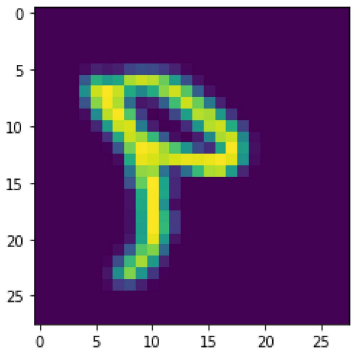
In [14]:

```
# Fit the transformation to the training dataset
train_datagenerator.fit(x_train)
```

In [15]:

```
# showing sample of a transformation
plt.imshow(train_datagenerator.random_transform(x_train[4]))
```

Out[15]: <matplotlib.image.AxesImage at 0x16ba21733d0>



Step 4: Model Architecture

In [16]:

```
# Define the CNN architecture

# Conv2D(filters=32,kernel_size=(3, 3) means use 32 filters (also called kernels) of size 3x3

# input_shape=(28, 28, 1) means the input data is a 3D tensor with dimensions 28x28x1.(height x width x number
    # The first dimension of the tensor (28) is the height of the image.
    #The second dimension of the tensor (28) is the width of the image.
    #The third dimension of the tensor (1) is the number of channels in the image (Black, White colors of the

# The MaxPooling2D(2, 2) performs a pooling operation where the (2,2) argument indicates the size of the pool
    # It reduces the spatial size of the feature maps and prevent overfitting
    # The output of the convolutional and pooling layers is a 3D tensor;

# The Flatten() layer takes 3D tensor (height, width, channels) and reshapes it into a 1D array;
    # The output of the convolutional and pooling layers is a 3D tensor;
    # 1D array is required by the fully connected (dense) layers;
    # Flatten() later enables the transition from the convolutional and pooling layers to the fully connected


cnn_model = Sequential()

cnn_model.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),activation = 'relu',padding = 'same')
cnn_model.add(MaxPooling2D(pool_size=(2,2)))

cnn_model.add(Conv2D(filters=64,kernel_size=(3,3),activation = 'relu'))
cnn_model.add(MaxPooling2D(pool_size=(2,2)))

cnn_model.add(Conv2D(filters=128,kernel_size=(3,3),activation = 'relu'))
cnn_model.add(MaxPooling2D(pool_size=(2,2)))

cnn_model.add(Flatten())
cnn_model.add(Dense(128,activation = 'relu'))
cnn_model.add(Dropout(0.5))
cnn_model.add(Dense(10,activation = 'softmax'))
```

In [17]:

```
 # print the summary of the model constructed
cnn_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 32)        320

 max_pooling2d (MaxPooling2D  (None, 14, 14, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 12, 12, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 128)         73856

 max_pooling2d_2 (MaxPooling  (None, 2, 2, 128)        0
 2D)

 flatten (Flatten)           (None, 512)               0

 dense (Dense)               (None, 128)               65664

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 159,626
Trainable params: 159,626
Non-trainable params: 0
_________________________________________________________________
```

Step 5: Compilation stage

In [18]:

```
# Compile the model with categorical cross-entropy loss and Adam optimizer
#categorical_crossentropy - loss function during model training
#accuracy - Evaluation measure

# learning_rate=0.001 specifies the step size of the updates to the weights during training.
# momentum=0.9 is a parameter that helps the optimizer to accelerate in the direction of the gradient and damp
# Higher values of momentum allow the optimizer to move more smoothly towards the global minimum
  # and reduce the chance of getting stuck in local minima.

sgd = SGD(learning_rate=0.001, momentum=0.9)
cnn_model.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
```

Step 6: Training and Evaluation stage

- **OPTIMIZER: SGD**
- The following parameters will be kept constant : **Convolutional block = 3, Learning_rate = 0.001, Batch_size = 32** and **Epochs = 20**

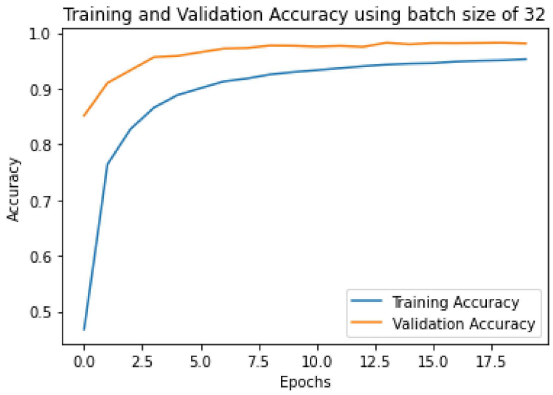
In [19]:

```
# Train the model with 20 epochs and batch size of 32
# to_categorical () converts integer class labels into one-hot encoded vectors;
# also used to convert predicted scores back into class labels for evaluation in classification problems.

# batch_size=32: the training dataset is put into batches of size 32.

history = cnn_model.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_size=32)
         epochs=20, validation_data=(x_test, keras.utils.to_categorical(y_test)))
```

```
Epoch 1/20
1875/1875 [==============================] - 61s 32ms/step - loss: 1.5391 - accuracy: 0.4680 - val_loss: 0.4651 -
val_accuracy: 0.8518
Epoch 2/20
1875/1875 [==============================] - 59s 31ms/step - loss: 0.7021 - accuracy: 0.7642 - val_loss: 0.2708 -
val_accuracy: 0.9102
Epoch 3/20
1875/1875 [==============================] - 56s 30ms/step - loss: 0.5288 - accuracy: 0.8276 - val_loss: 0.2033 -
val_accuracy: 0.9333
Epoch 4/20
1875/1875 [==============================] - 56s 30ms/step - loss: 0.4263 - accuracy: 0.8661 - val_loss: 0.1381 -
val_accuracy: 0.9567
Epoch 5/20
1875/1875 [==============================] - 55s 29ms/step - loss: 0.3579 - accuracy: 0.8882 - val_loss: 0.1286 -
val_accuracy: 0.9589
Epoch 6/20
1875/1875 [==============================] - 53s 28ms/step - loss: 0.3168 - accuracy: 0.9007 - val_loss: 0.1029 -
val_accuracy: 0.9656
Epoch 7/20
1875/1875 [==============================] - 53s 28ms/step - loss: 0.2824 - accuracy: 0.9129 - val_loss: 0.0834 -
val_accuracy: 0.9720
Epoch 8/20
1875/1875 [==============================] - 53s 28ms/step - loss: 0.2657 - accuracy: 0.9182 - val_loss: 0.0802 -
val_accuracy: 0.9730
Epoch 9/20
1875/1875 [==============================] - 53s 28ms/step - loss: 0.2443 - accuracy: 0.9255 - val_loss: 0.0654 -
val_accuracy: 0.9775
Epoch 10/20
1875/1875 [==============================] - 52s 28ms/step - loss: 0.2324 - accuracy: 0.9299 - val_loss: 0.0693 -
val_accuracy: 0.9771
Epoch 11/20
1875/1875 [==============================] - 54s 29ms/step - loss: 0.2181 - accuracy: 0.9332 - val_loss: 0.0747 -
val_accuracy: 0.9755
Epoch 12/20
1875/1875 [==============================] - 49s 26ms/step - loss: 0.2052 - accuracy: 0.9369 - val_loss: 0.0658 -
val_accuracy: 0.9769
Epoch 13/20
1875/1875 [==============================] - 52s 28ms/step - loss: 0.1961 - accuracy: 0.9403 - val_loss: 0.0763 -
val_accuracy: 0.9751
Epoch 14/20
1875/1875 [==============================] - 51s 27ms/step - loss: 0.1846 - accuracy: 0.9434 - val_loss: 0.0540 -
val_accuracy: 0.9824
Epoch 15/20
1875/1875 [==============================] - 49s 26ms/step - loss: 0.1846 - accuracy: 0.9450 - val_loss: 0.0605 -
val_accuracy: 0.9795
Epoch 16/20
1875/1875 [==============================] - 49s 26ms/step - loss: 0.1770 - accuracy: 0.9459 - val_loss: 0.0571 -
val_accuracy: 0.9818
Epoch 17/20
1875/1875 [==============================] - 49s 26ms/step - loss: 0.1717 - accuracy: 0.9487 - val_loss: 0.0557 -
val_accuracy: 0.9816
Epoch 18/20
1875/1875 [==============================] - 49s 26ms/step - loss: 0.1628 - accuracy: 0.9502 - val_loss: 0.0509 -
val_accuracy: 0.9820
Epoch 19/20
1875/1875 [==============================] - 50s 26ms/step - loss: 0.1589 - accuracy: 0.9512 - val_loss: 0.0518 -
val_accuracy: 0.9825
Epoch 20/20
1875/1875 [==============================] - 49s 26ms/step - loss: 0.1549 - accuracy: 0.9530 - val_loss: 0.0524 -
val_accuracy: 0.9810
```

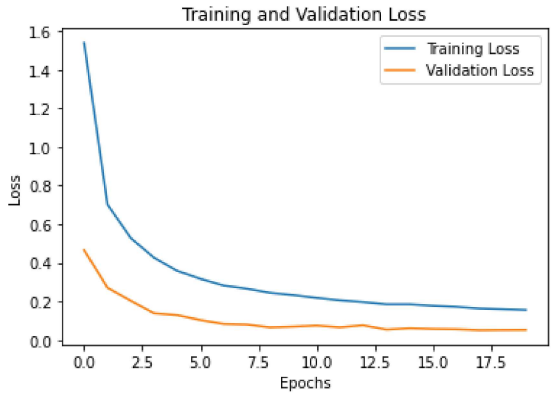
In [20]:

```
## checking for overfitting
# Plot training and validation accuracy

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy using batch size of 32')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [21]:

```
## checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



Step 7: Test the model (Make predictions)

In [22]:

```
# Predict the classes of the test set

#np.argmax(model.predict(x_test), axis=-1),
#this means that the argmax operation will be applied to the output predictions of the model along the last di
#this is the class probabilities for each test instance.
# argmax returns the class with the largest predicted probability.

y_pred = np.argmax(cnn_model.predict(x_test), axis=-1)
```

```
313/313 [==============================] - 2s 6ms/step
```

Step 8: Visualisation

In [23]:
```
from sklearn.metrics import classification_report # for visualisation

cnn_classification_report = classification_report(y_test, y_pred,output_dict=True)

print(classification_report(y_test, y_pred))
```

```
<frozen importlib._bootstrap>:228: RuntimeWarning: scipy._lib.messagestream.MessageStream size changed, may indic
ate binary incompatibility. Expected 56 from C header, got 64 from PyObject
```

```
              precision    recall  f1-score   support

           0       1.00      0.99      0.99       980
           1       1.00      0.99      0.99      1135
           2       0.94      0.98      0.96      1032
           3       0.99      0.98      0.99      1010
           4       0.98      0.99      0.98       982
           5       0.98      0.96      0.97       892
           6       0.98      0.97      0.98       958
           7       0.98      0.97      0.98      1028
           8       0.99      0.99      0.99       974
           9       0.98      0.98      0.98      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

In [24]:
```
data_report = pd.DataFrame(cnn_classification_report).transpose()
data_report
```

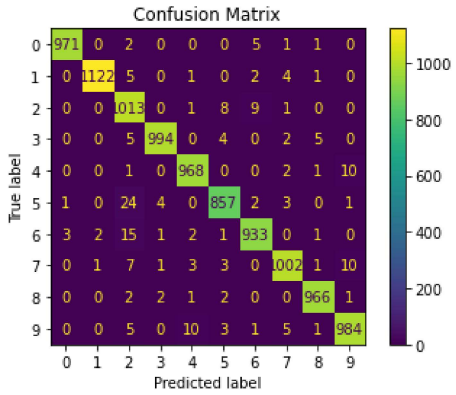
Out[24]:

	precision	recall	f1-score	support
0	0.995897	0.990816	0.993350	980.000
1	0.997333	0.988546	0.992920	1135.000
2	0.938832	0.981589	0.959735	1032.000
3	0.992016	0.984158	0.988072	1010.000
4	0.981744	0.985743	0.983740	982.000
5	0.976082	0.960762	0.968362	892.000
6	0.980042	0.973904	0.976963	958.000
7	0.982353	0.974708	0.978516	1028.000
8	0.988741	0.991786	0.990261	974.000
9	0.978131	0.975223	0.976675	1009.000
accuracy	0.981000	0.981000	0.981000	0.981
macro avg	0.981117	0.980724	0.980859	10000.000
weighted avg	0.981221	0.981000	0.981048	10000.000

In [25]:

```
# Print the confusion matrix
from sklearn.metrics import ConfusionMatrixDisplay


plt.figure(dpi=200, figsize=(10,15))
ConfusionMatrixDisplay.from_predictions(y_test,y_pred)
plt.title('Confusion Matrix')
plt.show()
```

```
<Figure size 2000x3000 with 0 Axes>
```



- **a.How did the use of different regularisation methods affect the performance of your CNN model?**
  - Regularisation helps us improve the generalisation of our model on unseen data. Some regularization methods are:
    - Dropout
    - Early Stopping
    - L1 and L2
    - Data Augmentation
    - Batch Normaliztion

Early Stopping Regularisation Method with the same optimizer and hyperparameters

Convolutional block = 3, Learning_rate = 0.001, Epoch = 20, Batch size = 32

In [26]:

```
cnn_model2 = Sequential()

cnn_model2.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),activation = 'relu',padding = 'same'
cnn_model2.add(MaxPooling2D(pool_size=(2,2)))

cnn_model2.add(Conv2D(filters=64,kernel_size=(3,3),activation = 'relu'))
cnn_model2.add(MaxPooling2D(pool_size=(2,2)))

cnn_model2.add(Conv2D(filters=128,kernel_size=(3,3),activation = 'relu'))
cnn_model2.add(MaxPooling2D(pool_size=(2,2)))

cnn_model2.add(Flatten())
cnn_model2.add(Dense(128,activation = 'relu'))
cnn_model2.add(Dropout(0.5))
cnn_model2.add(Dense(10,activation = 'softmax'))

#compiling the model
sgd = SGD(learning_rate=0.001, momentum=0.9)
cnn_model2.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
```

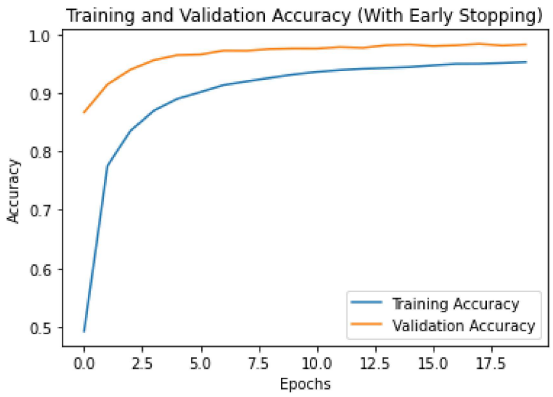
In [27]:

```
#Training the model
# Callbacks already imported above

earlystopping = callbacks.EarlyStopping(monitor="val_loss",mode="min", patience=5,restore_best_weights=True,v

history2 = cnn_model2.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_size=3
          epochs=20, validation_data=(x_test, keras.utils.to_categorical(y_test)),callbacks=[earlystopping])
```

```
Epoch 1/20
1875/1875 [==============================] - 50s 26ms/step - loss: 1.4684 - accuracy: 0.4925 - val_loss: 0.4177 -
val_accuracy: 0.8670
Epoch 2/20
1875/1875 [==============================] - 51s 27ms/step - loss: 0.6758 - accuracy: 0.7753 - val_loss: 0.2551 -
val_accuracy: 0.9149
Epoch 3/20
1875/1875 [==============================] - 53s 28ms/step - loss: 0.5018 - accuracy: 0.8357 - val_loss: 0.1865 -
val_accuracy: 0.9401
Epoch 4/20
1875/1875 [==============================] - 54s 29ms/step - loss: 0.4100 - accuracy: 0.8700 - val_loss: 0.1287 -
val_accuracy: 0.9563
Epoch 5/20
1875/1875 [==============================] - 53s 28ms/step - loss: 0.3500 - accuracy: 0.8900 - val_loss: 0.1044 -
val_accuracy: 0.9649
Epoch 6/20
1875/1875 [==============================] - 53s 28ms/step - loss: 0.3166 - accuracy: 0.9017 - val_loss: 0.0995 -
val_accuracy: 0.9660
Epoch 7/20
1875/1875 [==============================] - 52s 28ms/step - loss: 0.2795 - accuracy: 0.9138 - val_loss: 0.0823 -
val_accuracy: 0.9727
Epoch 8/20
1875/1875 [==============================] - 54s 29ms/step - loss: 0.2606 - accuracy: 0.9201 - val_loss: 0.0819 -
val_accuracy: 0.9725
Epoch 9/20
1875/1875 [==============================] - 51s 27ms/step - loss: 0.2402 - accuracy: 0.9260 - val_loss: 0.0714 -
val_accuracy: 0.9754
Epoch 10/20
1875/1875 [==============================] - 52s 28ms/step - loss: 0.2266 - accuracy: 0.9316 - val_loss: 0.0660 -
val_accuracy: 0.9763
Epoch 11/20
1875/1875 [==============================] - 52s 28ms/step - loss: 0.2106 - accuracy: 0.9361 - val_loss: 0.0640 -
val_accuracy: 0.9762
Epoch 12/20
1875/1875 [==============================] - 54s 29ms/step - loss: 0.1997 - accuracy: 0.9392 - val_loss: 0.0629 -
val_accuracy: 0.9788
Epoch 13/20
1875/1875 [==============================] - 52s 28ms/step - loss: 0.1911 - accuracy: 0.9415 - val_loss: 0.0705 -
val_accuracy: 0.9775
Epoch 14/20
1875/1875 [==============================] - 54s 29ms/step - loss: 0.1858 - accuracy: 0.9429 - val_loss: 0.0550 -
val_accuracy: 0.9820
Epoch 15/20
1875/1875 [==============================] - 61s 32ms/step - loss: 0.1799 - accuracy: 0.9444 - val_loss: 0.0506 -
val_accuracy: 0.9832
Epoch 16/20
1875/1875 [==============================] - 58s 31ms/step - loss: 0.1714 - accuracy: 0.9473 - val_loss: 0.0565 -
val_accuracy: 0.9806
Epoch 17/20
1875/1875 [==============================] - 50s 27ms/step - loss: 0.1664 - accuracy: 0.9498 - val_loss: 0.0541 -
val_accuracy: 0.9819
Epoch 18/20
1875/1875 [==============================] - 52s 28ms/step - loss: 0.1661 - accuracy: 0.9500 - val_loss: 0.0458 -
val_accuracy: 0.9842
Epoch 19/20
1875/1875 [==============================] - 53s 28ms/step - loss: 0.1570 - accuracy: 0.9514 - val_loss: 0.0561 -
val_accuracy: 0.9815
Epoch 20/20
1875/1875 [==============================] - 50s 27ms/step - loss: 0.1515 - accuracy: 0.9532 - val_loss: 0.0501 -
val_accuracy: 0.9833
```

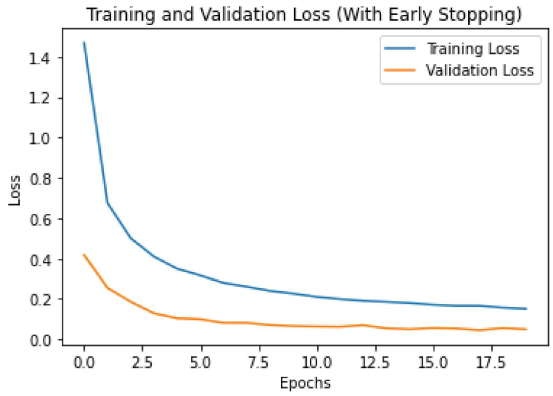
In [28]:

```
## checking for overfitting

# Plot training and validation accuracy

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history2.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history2.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With Early Stopping)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [29]:

```
# checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history2.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history2.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss (With Early Stopping)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [30]:

```
#make predictions
y_pred2 = np.argmax(cnn_model2.predict(x_test), axis=-1)
```

```
313/313 [==============================] - 3s 8ms/step
```

In [31]:

```
#Visualize the result
cnn_classification_report2 = classification_report(y_test, y_pred2,output_dict=True)

print(classification_report(y_test, y_pred2))
```

```
              precision    recall  f1-score   support

           0       0.99      1.00      0.99       980
           1       0.99      0.99      0.99      1135
           2       0.97      0.96      0.96      1032
           3       1.00      0.98      0.99      1010
           4       0.98      0.99      0.99       982
           5       0.97      0.97      0.97       892
           6       0.98      0.97      0.98       958
           7       0.97      0.99      0.98      1028
           8       0.99      0.99      0.99       974
           9       0.99      0.97      0.98      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

In [32]:

```
df_report2 = pd.DataFrame(cnn_classification_report2).transpose()
df_report2
```

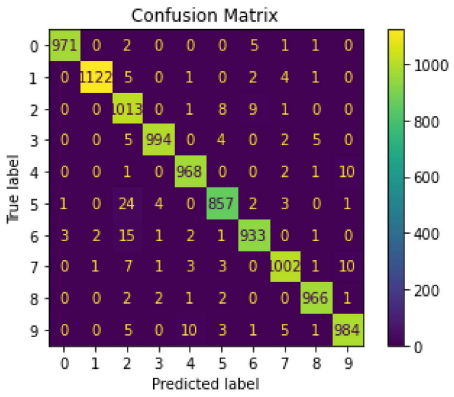
Out[32]:

	precision	recall	f1-score	support
0	0.986882	0.997959	0.992390	980.0000
1	0.993838	0.994714	0.994276	1135.0000
2	0.969638	0.959302	0.964442	1032.0000
3	0.996991	0.984158	0.990533	1010.0000
4	0.977934	0.992872	0.985346	982.0000
5	0.970950	0.974215	0.972580	892.0000
6	0.980021	0.972860	0.976427	958.0000
7	0.972355	0.992218	0.982186	1028.0000
8	0.989765	0.992813	0.991287	974.0000
9	0.992901	0.970268	0.981454	1009.0000
accuracy	0.983300	0.983300	0.983300	0.9833
macro avg	0.983127	0.983138	0.983092	10000.0000
weighted avg	0.983350	0.983300	0.983284	10000.0000

In [33]:

```
# Print the confusion matrix
from sklearn.metrics import ConfusionMatrixDisplay

plt.figure(dpi=200, figsize=(10,15))
ConfusionMatrixDisplay.from_predictions(y_test,y_pred)
plt.title('Confusion Matrix')
plt.show()
```

```
<Figure size 2000x3000 with 0 Axes>
```



Early Stopping Regularisation Method with the same optimizer and Learning Rate = 0.001, Epoch at 20 and batch size of 64

- This is to compare the validation loss for optimal performance

In [34]:

```
cnn_model2_1 = Sequential()

cnn_model2_1.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),activation = 'relu',padding = 'same
cnn_model2_1.add(MaxPooling2D(pool_size=(2,2)))

cnn_model2_1.add(Conv2D(filters=64,kernel_size=(3,3),activation = 'relu'))
cnn_model2_1.add(MaxPooling2D(pool_size=(2,2)))

cnn_model2_1.add(Conv2D(filters=128,kernel_size=(3,3),activation = 'relu'))
cnn_model2_1.add(MaxPooling2D(pool_size=(2,2)))

cnn_model2_1.add(Flatten())
cnn_model2_1.add(Dense(128,activation = 'relu'))
cnn_model2_1.add(Dropout(0.5))
cnn_model2_1.add(Dense(10,activation = 'softmax'))

#compiling the model
sgd = SGD(learning_rate=0.001, momentum=0.9)
cnn_model2_1.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
```

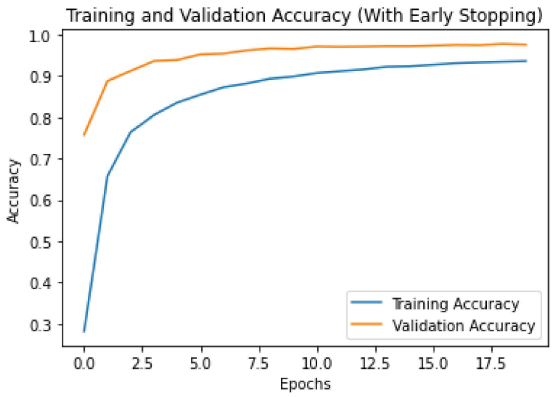
In [35]:

```
#Training the model
# Callbacks already imported above

earlystopping = callbacks.EarlyStopping(monitor="val_loss",mode="min", patience=5,restore_best_weights=True,v

history2_1 = cnn_model2_1.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_si
          epochs=20, validation_data=(x_test, keras.utils.to_categorical(y_test)),callbacks=[earlystopping])
```

```
Epoch 1/20
938/938 [==============================] - 47s 50ms/step - loss: 2.0772 - accuracy: 0.2821 - val_loss: 0.9581 - v
al_accuracy: 0.7577
Epoch 2/20
938/938 [==============================] - 52s 56ms/step - loss: 1.0123 - accuracy: 0.6576 - val_loss: 0.3714 - v
al_accuracy: 0.8878
Epoch 3/20
938/938 [==============================] - 46s 49ms/step - loss: 0.7068 - accuracy: 0.7640 - val_loss: 0.2789 - v
al_accuracy: 0.9120
Epoch 4/20
938/938 [==============================] - 47s 50ms/step - loss: 0.5878 - accuracy: 0.8054 - val_loss: 0.2125 - v
al_accuracy: 0.9361
Epoch 5/20
938/938 [==============================] - 46s 49ms/step - loss: 0.5057 - accuracy: 0.8353 - val_loss: 0.1879 - v
al_accuracy: 0.9386
Epoch 6/20
938/938 [==============================] - 47s 50ms/step - loss: 0.4500 - accuracy: 0.8547 - val_loss: 0.1538 - v
al_accuracy: 0.9522
Epoch 7/20
938/938 [==============================] - 47s 50ms/step - loss: 0.4017 - accuracy: 0.8728 - val_loss: 0.1432 - v
al_accuracy: 0.9543
Epoch 8/20
938/938 [==============================] - 47s 50ms/step - loss: 0.3751 - accuracy: 0.8819 - val_loss: 0.1204 - v
al_accuracy: 0.9619
Epoch 9/20
938/938 [==============================] - 48s 51ms/step - loss: 0.3426 - accuracy: 0.8934 - val_loss: 0.1080 - v
al_accuracy: 0.9667
Epoch 10/20
938/938 [==============================] - 51s 55ms/step - loss: 0.3246 - accuracy: 0.8985 - val_loss: 0.1051 - v
al_accuracy: 0.9655
Epoch 11/20
938/938 [==============================] - 48s 51ms/step - loss: 0.3014 - accuracy: 0.9068 - val_loss: 0.0900 - v
al_accuracy: 0.9712
Epoch 12/20
938/938 [==============================] - 48s 51ms/step - loss: 0.2846 - accuracy: 0.9114 - val_loss: 0.0866 - v
al_accuracy: 0.9705
Epoch 13/20
938/938 [==============================] - 49s 52ms/step - loss: 0.2735 - accuracy: 0.9155 - val_loss: 0.0850 - v
al_accuracy: 0.9711
Epoch 14/20
938/938 [==============================] - 48s 51ms/step - loss: 0.2571 - accuracy: 0.9220 - val_loss: 0.0847 - v
al_accuracy: 0.9721
Epoch 15/20
938/938 [==============================] - 47s 51ms/step - loss: 0.2514 - accuracy: 0.9232 - val_loss: 0.0808 - v
al_accuracy: 0.9721
Epoch 16/20
938/938 [==============================] - 48s 51ms/step - loss: 0.2382 - accuracy: 0.9270 - val_loss: 0.0735 - v
al_accuracy: 0.9736
Epoch 17/20
938/938 [==============================] - 51s 55ms/step - loss: 0.2272 - accuracy: 0.9308 - val_loss: 0.0691 - v
al_accuracy: 0.9754
Epoch 18/20
938/938 [==============================] - 51s 54ms/step - loss: 0.2206 - accuracy: 0.9329 - val_loss: 0.0737 - v
al_accuracy: 0.9748
Epoch 19/20
938/938 [==============================] - 49s 52ms/step - loss: 0.2172 - accuracy: 0.9344 - val_loss: 0.0619 - v
al_accuracy: 0.9777
Epoch 20/20
938/938 [==============================] - 48s 51ms/step - loss: 0.2114 - accuracy: 0.9359 - val_loss: 0.0672 - v
al_accuracy: 0.9759
```

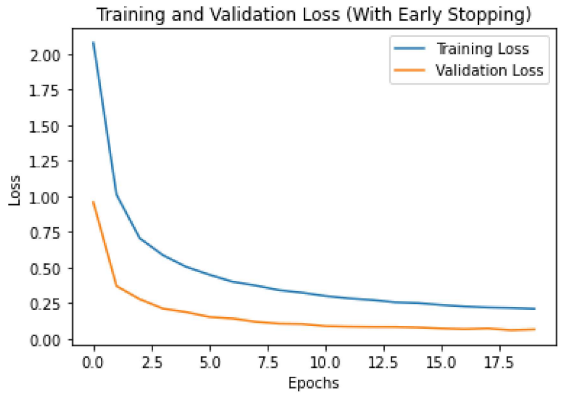
In [36]:

```
## checking for overfitting

# Plot training and validation accuracy

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history2_1.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history2_1.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With Early Stopping)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [37]:

```
# checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history2_1.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history2_1.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss (With Early Stopping)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



Early Stopping Regularisation Method with the same optimizer, Learning Rate = 0.001, Epoch at 25 and batch size of 32

- This is to compare the validation loss for optimal performance

In [38]:

```
cnn_model2_2 = Sequential()

cnn_model2_2.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),activation = 'relu',padding = 'sam
cnn_model2_2.add(MaxPooling2D(pool_size=(2,2)))

cnn_model2_2.add(Conv2D(filters=64,kernel_size=(3,3),activation = 'relu'))
cnn_model2_2.add(MaxPooling2D(pool_size=(2,2)))

cnn_model2_2.add(Conv2D(filters=128,kernel_size=(3,3),activation = 'relu'))
cnn_model2_2.add(MaxPooling2D(pool_size=(2,2)))

cnn_model2_2.add(Flatten())
cnn_model2_2.add(Dense(128,activation = 'relu'))
cnn_model2_2.add(Dropout(0.5))
cnn_model2_2.add(Dense(10,activation = 'softmax'))

#compiling the model
sgd = SGD(learning_rate=0.001, momentum=0.9)
cnn_model2_2.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
```

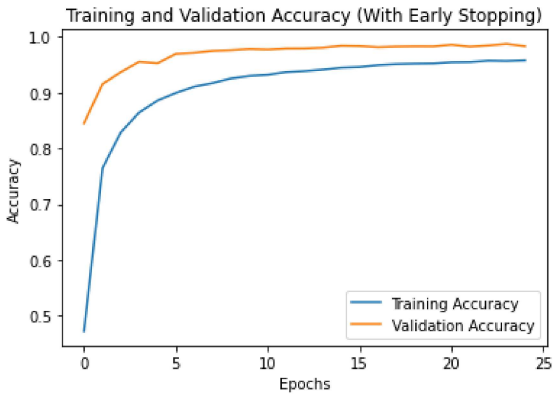
In [39]:

```
#Training the model
# Callbacks already imported above

earlystopping = callbacks.EarlyStopping(monitor="val_loss",mode="min", patience=5,restore_best_weights=True,v

history2_2 = cnn_model2_2.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_si
          epochs=25, validation_data=(x_test, keras.utils.to_categorical(y_test)),callbacks=[earlystopping])
```

```
Epoch 1/25
1875/1875 [==============================] - 50s 26ms/step - loss: 1.5261 - accuracy: 0.4725 - val_loss: 0.4626 -
val_accuracy: 0.8451
Epoch 2/25
1875/1875 [==============================] - 55s 29ms/step - loss: 0.7106 - accuracy: 0.7645 - val_loss: 0.2764 -
val_accuracy: 0.9153
Epoch 3/25
1875/1875 [==============================] - 56s 30ms/step - loss: 0.5247 - accuracy: 0.8287 - val_loss: 0.1861 -
val_accuracy: 0.9365
Epoch 4/25
1875/1875 [==============================] - 53s 28ms/step - loss: 0.4241 - accuracy: 0.8643 - val_loss: 0.1343 -
val_accuracy: 0.9552
Epoch 5/25
1875/1875 [==============================] - 55s 29ms/step - loss: 0.3639 - accuracy: 0.8858 - val_loss: 0.1404 -
val_accuracy: 0.9525
Epoch 6/25
1875/1875 [==============================] - 53s 28ms/step - loss: 0.3203 - accuracy: 0.8994 - val_loss: 0.0947 -
val_accuracy: 0.9695
Epoch 7/25
1875/1875 [==============================] - 53s 28ms/step - loss: 0.2865 - accuracy: 0.9107 - val_loss: 0.0829 -
val_accuracy: 0.9713
Epoch 8/25
1875/1875 [==============================] - 55s 29ms/step - loss: 0.2709 - accuracy: 0.9171 - val_loss: 0.0738 -
val_accuracy: 0.9748
Epoch 9/25
1875/1875 [==============================] - 56s 30ms/step - loss: 0.2443 - accuracy: 0.9254 - val_loss: 0.0705 -
val_accuracy: 0.9760
Epoch 10/25
1875/1875 [==============================] - 50s 27ms/step - loss: 0.2263 - accuracy: 0.9298 - val_loss: 0.0649 -
val_accuracy: 0.9781
Epoch 11/25
1875/1875 [==============================] - 62s 33ms/step - loss: 0.2226 - accuracy: 0.9318 - val_loss: 0.0715 -
val_accuracy: 0.9771
Epoch 12/25
1875/1875 [==============================] - 61s 32ms/step - loss: 0.2076 - accuracy: 0.9366 - val_loss: 0.0601 -
val_accuracy: 0.9790
Epoch 13/25
1875/1875 [==============================] - 63s 34ms/step - loss: 0.2029 - accuracy: 0.9384 - val_loss: 0.0592 -
val_accuracy: 0.9791
Epoch 14/25
1875/1875 [==============================] - 55s 29ms/step - loss: 0.1890 - accuracy: 0.9411 - val_loss: 0.0574 -
val_accuracy: 0.9806
Epoch 15/25
1875/1875 [==============================] - 56s 30ms/step - loss: 0.1829 - accuracy: 0.9446 - val_loss: 0.0518 -
val_accuracy: 0.9840
Epoch 16/25
1875/1875 [==============================] - 60s 32ms/step - loss: 0.1783 - accuracy: 0.9458 - val_loss: 0.0476 -
val_accuracy: 0.9835
Epoch 17/25
1875/1875 [==============================] - 61s 32ms/step - loss: 0.1676 - accuracy: 0.9492 - val_loss: 0.0538 -
val_accuracy: 0.9814
Epoch 18/25
1875/1875 [==============================] - 62s 33ms/step - loss: 0.1635 - accuracy: 0.9510 - val_loss: 0.0542 -
val_accuracy: 0.9825
Epoch 19/25
1875/1875 [==============================] - 59s 31ms/step - loss: 0.1585 - accuracy: 0.9517 - val_loss: 0.0543 -
val_accuracy: 0.9829
Epoch 20/25
1875/1875 [==============================] - 58s 31ms/step - loss: 0.1543 - accuracy: 0.9520 - val_loss: 0.0496 -
val_accuracy: 0.9828
Epoch 21/25
1875/1875 [==============================] - 56s 30ms/step - loss: 0.1488 - accuracy: 0.9543 - val_loss: 0.0416 -
val_accuracy: 0.9859
Epoch 22/25
1875/1875 [==============================] - 60s 32ms/step - loss: 0.1493 - accuracy: 0.9547 - val_loss: 0.0507 -
val_accuracy: 0.9824
Epoch 23/25
1875/1875 [==============================] - 60s 32ms/step - loss: 0.1437 - accuracy: 0.9571 - val_loss: 0.0452 -
val_accuracy: 0.9845
Epoch 24/25
1875/1875 [==============================] - 57s 30ms/step - loss: 0.1427 - accuracy: 0.9565 - val_loss: 0.0378 -
val_accuracy: 0.9873
Epoch 25/25
1875/1875 [==============================] - 58s 31ms/step - loss: 0.1392 - accuracy: 0.9577 - val_loss: 0.0469 -
val_accuracy: 0.9831
```

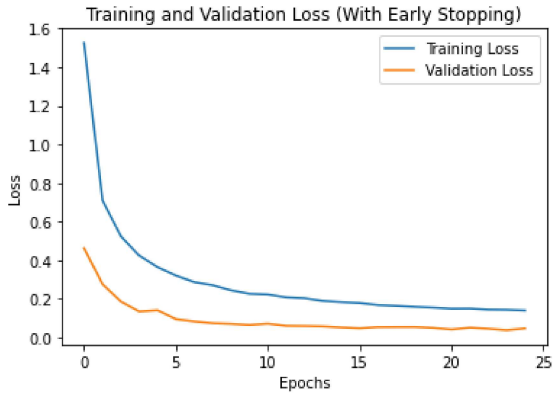
In [40]:

```
## checking for overfitting

# Plot training and validation accuracy

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history2_2.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history2_2.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With Early Stopping)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [41]:

```
# checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history2_2.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history2_2.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss (With Early Stopping)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



Early Stopping Regularisation Method with the same optimizer, Learning Rate = 0.001, Epoch at 25 and batch size of 64

- To compare performance

In [42]:

```
cnn_model2_3 = Sequential()

cnn_model2_3.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),activation = 'relu',padding = 'sam
cnn_model2_3.add(MaxPooling2D(pool_size=(2,2)))

cnn_model2_3.add(Conv2D(filters=64,kernel_size=(3,3),activation = 'relu'))
cnn_model2_3.add(MaxPooling2D(pool_size=(2,2)))

cnn_model2_3.add(Conv2D(filters=128,kernel_size=(3,3),activation = 'relu'))
cnn_model2_3.add(MaxPooling2D(pool_size=(2,2)))

cnn_model2_3.add(Flatten())
cnn_model2_3.add(Dense(128,activation = 'relu'))
cnn_model2_3.add(Dropout(0.5))
cnn_model2_3.add(Dense(10,activation = 'softmax'))

#compiling the model
sgd = SGD(learning_rate=0.001, momentum=0.9)
cnn_model2_3.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
```

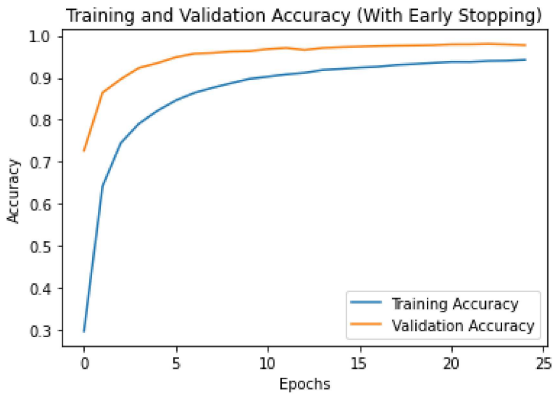
In [43]:

```
#Training the model
# Callbacks already imported above

earlystopping = callbacks.EarlyStopping(monitor="val_loss",mode="min", patience=5,restore_best_weights=True,v

history2_3 = cnn_model2_3.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_si
          epochs=25, validation_data=(x_test, keras.utils.to_categorical(y_test)),callbacks=[earlystopping])
```

```
Epoch 1/25
938/938 [==============================] - 46s 48ms/step - loss: 2.0115 - accuracy: 0.2977 - val_loss: 0.9175 - v
al_accuracy: 0.7268
Epoch 2/25
938/938 [==============================] - 52s 55ms/step - loss: 1.0517 - accuracy: 0.6417 - val_loss: 0.4320 - v
al_accuracy: 0.8646
Epoch 3/25
938/938 [==============================] - 52s 55ms/step - loss: 0.7552 - accuracy: 0.7448 - val_loss: 0.3158 - v
al_accuracy: 0.8966
Epoch 4/25
938/938 [==============================] - 55s 58ms/step - loss: 0.6327 - accuracy: 0.7915 - val_loss: 0.2491 - v
al_accuracy: 0.9235
Epoch 5/25
938/938 [==============================] - 53s 57ms/step - loss: 0.5505 - accuracy: 0.8215 - val_loss: 0.2066 - v
al_accuracy: 0.9348
Epoch 6/25
938/938 [==============================] - 51s 54ms/step - loss: 0.4866 - accuracy: 0.8462 - val_loss: 0.1713 - v
al_accuracy: 0.9488
Epoch 7/25
938/938 [==============================] - 61s 65ms/step - loss: 0.4287 - accuracy: 0.8641 - val_loss: 0.1398 - v
al_accuracy: 0.9573
Epoch 8/25
938/938 [==============================] - 55s 59ms/step - loss: 0.3925 - accuracy: 0.8763 - val_loss: 0.1240 - v
al_accuracy: 0.9590
Epoch 9/25
938/938 [==============================] - 52s 56ms/step - loss: 0.3601 - accuracy: 0.8871 - val_loss: 0.1156 - v
al_accuracy: 0.9623
Epoch 10/25
938/938 [==============================] - 48s 51ms/step - loss: 0.3329 - accuracy: 0.8974 - val_loss: 0.1037 - v
al_accuracy: 0.9633
Epoch 11/25
938/938 [==============================] - 50s 53ms/step - loss: 0.3133 - accuracy: 0.9024 - val_loss: 0.0925 - v
al_accuracy: 0.9681
Epoch 12/25
938/938 [==============================] - 49s 52ms/step - loss: 0.2962 - accuracy: 0.9078 - val_loss: 0.0873 - v
al_accuracy: 0.9710
Epoch 13/25
938/938 [==============================] - 49s 52ms/step - loss: 0.2854 - accuracy: 0.9115 - val_loss: 0.0996 - v
al_accuracy: 0.9660
Epoch 14/25
938/938 [==============================] - 55s 59ms/step - loss: 0.2689 - accuracy: 0.9184 - val_loss: 0.0851 - v
al_accuracy: 0.9710
Epoch 15/25
938/938 [==============================] - 51s 54ms/step - loss: 0.2576 - accuracy: 0.9210 - val_loss: 0.0791 - v
al_accuracy: 0.9728
Epoch 16/25
938/938 [==============================] - 48s 52ms/step - loss: 0.2477 - accuracy: 0.9240 - val_loss: 0.0764 - v
al_accuracy: 0.9741
Epoch 17/25
938/938 [==============================] - 51s 54ms/step - loss: 0.2403 - accuracy: 0.9261 - val_loss: 0.0693 - v
al_accuracy: 0.9755
Epoch 18/25
938/938 [==============================] - 54s 58ms/step - loss: 0.2306 - accuracy: 0.9302 - val_loss: 0.0681 - v
al_accuracy: 0.9764
Epoch 19/25
938/938 [==============================] - 47s 50ms/step - loss: 0.2222 - accuracy: 0.9325 - val_loss: 0.0697 - v
al_accuracy: 0.9769
Epoch 20/25
938/938 [==============================] - 46s 50ms/step - loss: 0.2092 - accuracy: 0.9351 - val_loss: 0.0609 - v
al_accuracy: 0.9777
Epoch 21/25
938/938 [==============================] - 48s 51ms/step - loss: 0.2040 - accuracy: 0.9377 - val_loss: 0.0580 - v
al_accuracy: 0.9794
Epoch 22/25
938/938 [==============================] - 50s 54ms/step - loss: 0.2060 - accuracy: 0.9375 - val_loss: 0.0608 - v
al_accuracy: 0.9795
Epoch 23/25
938/938 [==============================] - 47s 50ms/step - loss: 0.1963 - accuracy: 0.9398 - val_loss: 0.0561 - v
al_accuracy: 0.9805
Epoch 24/25
938/938 [==============================] - 47s 50ms/step - loss: 0.1952 - accuracy: 0.9403 - val_loss: 0.0593 - v
al_accuracy: 0.9793
Epoch 25/25
938/938 [==============================] - 47s 50ms/step - loss: 0.1896 - accuracy: 0.9426 - val_loss: 0.0677 - v
al_accuracy: 0.9777
```

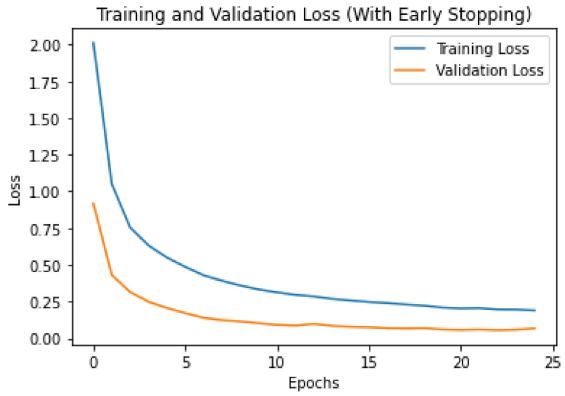
In [44]:

```
## checking for overfitting

# Plot training and validation accuracy

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history2_3.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history2_3.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With Early Stopping)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [45]:

```
# checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history2_3.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history2_3.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss (With Early Stopping)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



Observations:

- At epoch 20, batch size 32, the model had it's least validation loss of **0.0458 at the 18th epoch.**
- At epoch 20, batch size 64, the model had it's least validation loss of **0.0619 at the 19th epoch.**
- At epoch 25, batch size 32, the model had it's least validation loss of **0.0378 at the 19th epoch**. I will be making this *epoch and batch size constant* since it gave me the overall least validation loss.
- At epoch 25, batch size 64, the model had it's least validation loss of **0.0580 at the 21st epoch.**

Batch Normaliztion Regularisation Method with the same optimizer, Learning Rate = 0.001, epoch = 19, and batch size = 32 (from my constant)

- *NB: The above epoch and batch suze are my constant because of the optimal performance*

- I would examine the introduction of Batch Normalization into our model because it normalizes the matrix after it has been through a convolution layer. It reduces training time and ensures that the scale of each dimension remains the same.

In [46]:

```
cnn_model3 = Sequential()

cnn_model3.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),padding = 'same'))
cnn_model3.add(BatchNormalization())
cnn_model3.add(Activation('relu'))
cnn_model3.add(MaxPooling2D(pool_size=(2,2)))

cnn_model3.add(Conv2D(filters=64,kernel_size=(3,3)))
cnn_model3.add(BatchNormalization())
cnn_model3.add(Activation('relu'))
cnn_model3.add(MaxPooling2D(pool_size=(2,2)))

cnn_model3.add(Conv2D(filters=128,kernel_size=(3,3)))
cnn_model3.add(BatchNormalization())
cnn_model3.add(Activation('relu'))
cnn_model3.add(MaxPooling2D(pool_size=(2,2)))

cnn_model3.add(Flatten())
cnn_model3.add(BatchNormalization())
cnn_model3.add(Dense(128,activation = 'relu'))
cnn_model3.add(Dropout(0.5))
cnn_model3.add(Dense(10,activation = 'softmax'))

#compiling the model
sgd = SGD(learning_rate=0.001, momentum=0.9)
cnn_model3.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
```

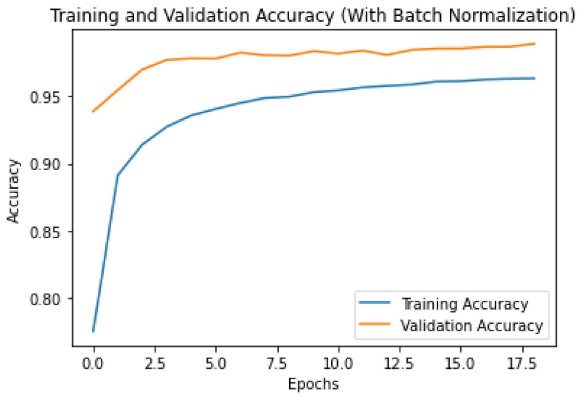
In [47]:

```
#Training and evaluating the model
history3 = cnn_model3.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_size=3
         epochs = 19, validation_data=(x_test, keras.utils.to_categorical(y_test)))
```

```
Epoch 1/19
1875/1875 [==============================] - 73s 38ms/step - loss: 0.6978 - accuracy: 0.7750 - val_loss: 0.1903 -
val_accuracy: 0.9389
Epoch 2/19
1875/1875 [==============================] - 76s 41ms/step - loss: 0.3503 - accuracy: 0.8912 - val_loss: 0.1595 -
val_accuracy: 0.9546
Epoch 3/19
1875/1875 [==============================] - 72s 38ms/step - loss: 0.2770 - accuracy: 0.9140 - val_loss: 0.0989 -
val_accuracy: 0.9700
Epoch 4/19
1875/1875 [==============================] - 80s 43ms/step - loss: 0.2345 - accuracy: 0.9273 - val_loss: 0.0712 -
val_accuracy: 0.9772
Epoch 5/19
1875/1875 [==============================] - 75s 40ms/step - loss: 0.2099 - accuracy: 0.9358 - val_loss: 0.0665 -
val_accuracy: 0.9783
Epoch 6/19
1875/1875 [==============================] - 73s 39ms/step - loss: 0.1946 - accuracy: 0.9406 - val_loss: 0.0721 -
val_accuracy: 0.9781
Epoch 7/19
1875/1875 [==============================] - 75s 40ms/step - loss: 0.1784 - accuracy: 0.9451 - val_loss: 0.0549 -
val_accuracy: 0.9826
Epoch 8/19
1875/1875 [==============================] - 79s 42ms/step - loss: 0.1688 - accuracy: 0.9487 - val_loss: 0.0611 -
val_accuracy: 0.9807
Epoch 9/19
1875/1875 [==============================] - 82s 44ms/step - loss: 0.1631 - accuracy: 0.9497 - val_loss: 0.0598 -
val_accuracy: 0.9804
Epoch 10/19
1875/1875 [==============================] - 107s 57ms/step - loss: 0.1511 - accuracy: 0.9530 - val_loss: 0.0564
- val_accuracy: 0.9837
Epoch 11/19
1875/1875 [==============================] - 93s 50ms/step - loss: 0.1474 - accuracy: 0.9543 - val_loss: 0.0601 -
val_accuracy: 0.9819
Epoch 12/19
1875/1875 [==============================] - 81s 43ms/step - loss: 0.1430 - accuracy: 0.9566 - val_loss: 0.0508 -
val_accuracy: 0.9841
Epoch 13/19
1875/1875 [==============================] - 83s 44ms/step - loss: 0.1405 - accuracy: 0.9578 - val_loss: 0.0591 -
val_accuracy: 0.9809
Epoch 14/19
1875/1875 [==============================] - 79s 42ms/step - loss: 0.1361 - accuracy: 0.9587 - val_loss: 0.0471 -
val_accuracy: 0.9846
Epoch 15/19
1875/1875 [==============================] - 79s 42ms/step - loss: 0.1311 - accuracy: 0.9610 - val_loss: 0.0449 -
val_accuracy: 0.9856
Epoch 16/19
1875/1875 [==============================] - 78s 42ms/step - loss: 0.1291 - accuracy: 0.9612 - val_loss: 0.0467 -
val_accuracy: 0.9856
Epoch 17/19
1875/1875 [==============================] - 82s 43ms/step - loss: 0.1253 - accuracy: 0.9625 - val_loss: 0.0442 -
val_accuracy: 0.9868
Epoch 18/19
1875/1875 [==============================] - 78s 42ms/step - loss: 0.1210 - accuracy: 0.9631 - val_loss: 0.0425 -
val_accuracy: 0.9869
Epoch 19/19
1875/1875 [==============================] - 83s 44ms/step - loss: 0.1173 - accuracy: 0.9633 - val_loss: 0.0374 -
val_accuracy: 0.9891
```

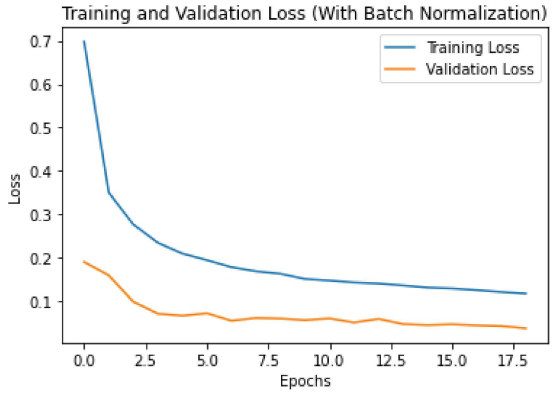
In [48]:

```
## checking for overfitting
'''
# Plot training and validation accuracy

'''

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history3.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history3.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With Batch Normalization)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [49]:

```
## checking for overfitting
'''
# Plot training loss against validation loss

'''
#Plots the training loss as a line with label "Training Loss".
plt.plot(history3.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history3.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss (With Batch Normalization)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [50]:

```
#make predictions
y_pred3 = np.argmax(cnn_model3.predict(x_test), axis=-1)
```

```
313/313 [==============================] - 3s 10ms/step
```

In [51]:

```
#Visualize the result
from sklearn.metrics import classification_report
cnn_classification_report3 = classification_report(y_test, y_pred3,output_dict=True)

print(classification_report(y_test, y_pred3))
```

```
              precision    recall  f1-score   support

           0       1.00      0.99      0.99       980
           1       1.00      0.99      0.99      1135
           2       0.98      0.98      0.98      1032
           3       0.99      1.00      0.99      1010
           4       0.99      1.00      0.99       982
           5       0.97      0.98      0.98       892
           6       0.99      0.99      0.99       958
           7       0.99      0.99      0.99      1028
           8       0.99      0.99      0.99       974
           9       0.99      0.98      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

In [52]:

```
df_report3 = pd.DataFrame(cnn_classification_report3).transpose()
df_report3
```

Out[52]:

	precision	recall	f1-score	support
0	0.997947	0.991837	0.994882	980.0000
1	0.998224	0.990308	0.994250	1135.0000
2	0.977756	0.979651	0.978703	1032.0000
3	0.989194	0.997030	0.993097	1010.0000
4	0.986908	0.997963	0.992405	982.0000
5	0.972315	0.984305	0.978273	892.0000
6	0.987474	0.987474	0.987474	958.0000
7	0.990253	0.988327	0.989289	1028.0000
8	0.993834	0.992813	0.993323	974.0000
9	0.994975	0.981169	0.988024	1009.0000
accuracy	0.989100	0.989100	0.989100	0.9891
macro avg	0.988888	0.989088	0.988972	10000.0000
weighted avg	0.989146	0.989100	0.989107	10000.0000

Observations:

- The model's accuracy increased by 0.182% while the validation loss reduced by 1.06% with the introduction of Batch Normalization.
- Subsequently, **Batch Normalization will be my constant Regularisation Method.**

L1 and L2 Regularisation Method with the same optimizer, Learning Rate = 0.001, epoch = 19, and batch size = 32

In [54]:

```
cnn_model4_2 = Sequential()

cnn_model4_2.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1), padding = 'same', kernel_regulari
cnn_model4_2.add(Activation('relu'))
cnn_model4_2.add(MaxPooling2D(pool_size=(2,2)))

cnn_model4_2.add(Conv2D(filters=64,kernel_size=(3,3), kernel_regularizer = regularizers.l1_l2(l1 = 0.01, l2 =
cnn_model4_2.add(Activation('relu'))
cnn_model4_2.add(MaxPooling2D(pool_size=(2,2)))

cnn_model4_2.add(Conv2D(filters=128,kernel_size=(3,3), kernel_regularizer = regularizers.l1_l2(l1 = 0.01, l2 =
cnn_model4_2.add(Activation('relu'))
cnn_model4_2.add(MaxPooling2D(pool_size=(2,2)))

cnn_model4_2.add(Flatten())
cnn_model4_2.add(Dense(128,activation = 'relu'))
cnn_model4_2.add(Dropout(0.5))
cnn_model4_2.add(Dense(10,activation = 'softmax'))

#compiling the model
sgd = SGD(learning_rate=0.001, momentum=0.9)
cnn_model4_2.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
```

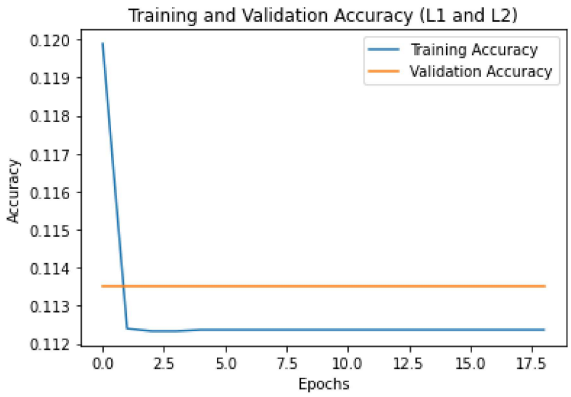
In [55]:

```
#Training and evaluating the model
history4_2 = cnn_model4_2.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_si
        epochs=19, validation_data=(x_test, keras.utils.to_categorical(y_test)))
```

```
Epoch 1/19
1875/1875 [==============================] - 54s 29ms/step - loss: 5.8304 - accuracy: 0.1199 - val_loss: 2.3116 -
val_accuracy: 0.1135
Epoch 2/19
1875/1875 [==============================] - 56s 30ms/step - loss: 2.3118 - accuracy: 0.1124 - val_loss: 2.3113 -
val_accuracy: 0.1135
Epoch 3/19
1875/1875 [==============================] - 53s 28ms/step - loss: 2.3115 - accuracy: 0.1123 - val_loss: 2.3111 -
val_accuracy: 0.1135
Epoch 4/19
1875/1875 [==============================] - 54s 29ms/step - loss: 2.3114 - accuracy: 0.1123 - val_loss: 2.3110 -
val_accuracy: 0.1135
Epoch 5/19
1875/1875 [==============================] - 54s 29ms/step - loss: 2.3112 - accuracy: 0.1124 - val_loss: 2.3109 -
val_accuracy: 0.1135
Epoch 6/19
1875/1875 [==============================] - 55s 29ms/step - loss: 2.3112 - accuracy: 0.1124 - val_loss: 2.3110 -
val_accuracy: 0.1135
Epoch 7/19
1875/1875 [==============================] - 56s 30ms/step - loss: 2.3112 - accuracy: 0.1124 - val_loss: 2.3109 -
val_accuracy: 0.1135
Epoch 8/19
1875/1875 [==============================] - 56s 30ms/step - loss: 2.3113 - accuracy: 0.1124 - val_loss: 2.3110 -
val_accuracy: 0.1135
Epoch 9/19
1875/1875 [==============================] - 55s 29ms/step - loss: 2.3113 - accuracy: 0.1124 - val_loss: 2.3110 -
val_accuracy: 0.1135
Epoch 10/19
1875/1875 [==============================] - 60s 32ms/step - loss: 2.3113 - accuracy: 0.1124 - val_loss: 2.3110 -
val_accuracy: 0.1135
Epoch 11/19
1875/1875 [==============================] - 60s 32ms/step - loss: 2.3113 - accuracy: 0.1124 - val_loss: 2.3111 -
val_accuracy: 0.1135
Epoch 12/19
1875/1875 [==============================] - 56s 30ms/step - loss: 2.3114 - accuracy: 0.1124 - val_loss: 2.3113 -
val_accuracy: 0.1135
Epoch 13/19
1875/1875 [==============================] - 58s 31ms/step - loss: 2.3116 - accuracy: 0.1124 - val_loss: 2.3114 -
val_accuracy: 0.1135
Epoch 14/19
1875/1875 [==============================] - 56s 30ms/step - loss: 2.3116 - accuracy: 0.1124 - val_loss: 2.3114 -
val_accuracy: 0.1135
Epoch 15/19
1875/1875 [==============================] - 55s 29ms/step - loss: 2.3116 - accuracy: 0.1124 - val_loss: 2.3113 -
val_accuracy: 0.1135
Epoch 16/19
1875/1875 [==============================] - 59s 31ms/step - loss: 2.3116 - accuracy: 0.1124 - val_loss: 2.3114 -
val_accuracy: 0.1135
Epoch 17/19
1875/1875 [==============================] - 54s 29ms/step - loss: 2.3117 - accuracy: 0.1124 - val_loss: 2.3115 -
val_accuracy: 0.1135
Epoch 18/19
1875/1875 [==============================] - 55s 29ms/step - loss: 2.3117 - accuracy: 0.1124 - val_loss: 2.3114 -
val_accuracy: 0.1135
Epoch 19/19
1875/1875 [==============================] - 55s 29ms/step - loss: 2.3117 - accuracy: 0.1124 - val_loss: 2.3113 -
val_accuracy: 0.1135
```

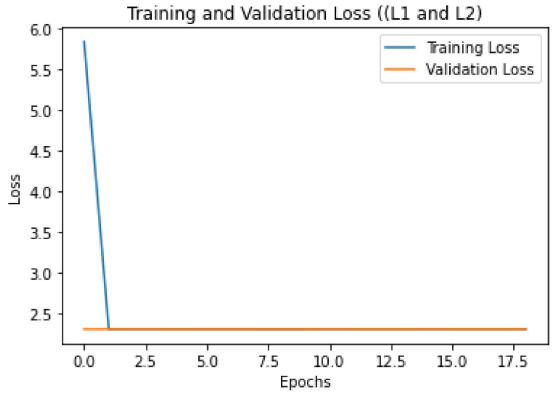
In [56]:

```
## checking for overfitting
'''
# Plot training and validation accuracy

'''

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history4_2.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history4_2.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (L1 and L2)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [57]:

```
## checking for overfitting
'''
# Plot training loss against validation loss

'''
#Plots the training loss as a line with label "Training Loss".
plt.plot(history4_2.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history4_2.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss ((L1 and L2)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [58]:

```
#make predictions
y_pred4_2 = np.argmax(cnn_model4_2.predict(x_test), axis=-1)
```

```
313/313 [==============================] - 3s 8ms/step
```

In [59]:

```python
#Visualize the result

from sklearn.metrics import classification_report
cnn_classification_report4_2 = classification_report(y_test, y_pred4_2,output_dict=True)

print(classification_report(y_test, y_pred4_2))
```

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00       980
           1       0.11      1.00      0.20      1135
           2       0.00      0.00      0.00      1032
           3       0.00      0.00      0.00      1010
           4       0.00      0.00      0.00       982
           5       0.00      0.00      0.00       892
           6       0.00      0.00      0.00       958
           7       0.00      0.00      0.00      1028
           8       0.00      0.00      0.00       974
           9       0.00      0.00      0.00      1009

    accuracy                           0.11     10000
   macro avg       0.01      0.10      0.02     10000
weighted avg       0.01      0.11      0.02     10000


C:\Users\Laptop\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1344: UndefinedMetricWarning: Prec
ision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` p
arameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\Laptop\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1344: UndefinedMetricWarning: Prec
ision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` p
arameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\Laptop\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1344: UndefinedMetricWarning: Prec
ision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` p
arameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\Laptop\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1344: UndefinedMetricWarning: Prec
ision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` p
arameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\Laptop\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1344: UndefinedMetricWarning: Prec
ision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` p
arameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\Laptop\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1344: UndefinedMetricWarning: Prec
ision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` p
arameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

In [60]:

```python
df_report4_2 = pd.DataFrame(cnn_classification_report4_2).transpose()
df_report4_2
```

Out[60]:

	precision	recall	f1-score	support
0	0.000000	0.0000	0.000000	980.0000
1	0.113500	1.0000	0.203862	1135.0000
2	0.000000	0.0000	0.000000	1032.0000
3	0.000000	0.0000	0.000000	1010.0000
4	0.000000	0.0000	0.000000	982.0000
5	0.000000	0.0000	0.000000	892.0000
6	0.000000	0.0000	0.000000	958.0000
7	0.000000	0.0000	0.000000	1028.0000
8	0.000000	0.0000	0.000000	974.0000
9	0.000000	0.0000	0.000000	1009.0000
accuracy	0.113500	0.1135	0.113500	0.1135
macro avg	0.011350	0.1000	0.020386	10000.0000
weighted avg	0.012882	0.1135	0.023138	10000.0000

Observations:

- The model's accuracy was low with L1 and L2 regularisation.
- This regularisation will not be used as it performed poorly.

Investigating the performance of the model with different optimizers

- **OPTIMIZER: ADAM**
- The following parameters will be kept constant : **Learning_rate = 0.001**, **Batch_size = 32** and **Epochs = 19**

In [61]:

```
cnn_model5 = Sequential()

cnn_model5.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),activation = 'relu',padding = 'same'
cnn_model5.add(MaxPooling2D(pool_size=(2,2)))

cnn_model5.add(Conv2D(filters=64,kernel_size=(3,3),activation = 'relu'))
cnn_model5.add(MaxPooling2D(pool_size=(2,2)))

cnn_model5.add(Conv2D(filters=128,kernel_size=(3,3),activation = 'relu'))
cnn_model5.add(MaxPooling2D(pool_size=(2,2)))

cnn_model5.add(Flatten())
cnn_model5.add(BatchNormalization())
cnn_model5.add(Dense(128,activation = 'relu'))
cnn_model5.add(Dropout(0.5))
cnn_model5.add(Dense(10,activation = 'softmax'))

#compiling the model
cnn_model5.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy']
```

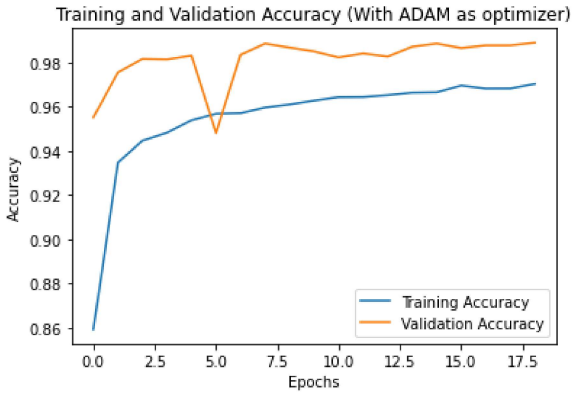
In [62]:

```
#Training and evaluating the model
history5 = cnn_model5.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_size=3
         epochs=19, validation_data=(x_test, keras.utils.to_categorical(y_test)))
```

```
Epoch 1/19
1875/1875 [==============================] - 67s 35ms/step - loss: 0.4374 - accuracy: 0.8594 - val_loss: 0.1598 -
val_accuracy: 0.9551
Epoch 2/19
1875/1875 [==============================] - 62s 33ms/step - loss: 0.2145 - accuracy: 0.9347 - val_loss: 0.0847 -
val_accuracy: 0.9754
Epoch 3/19
1875/1875 [==============================] - 61s 33ms/step - loss: 0.1858 - accuracy: 0.9445 - val_loss: 0.0653 -
val_accuracy: 0.9815
Epoch 4/19
1875/1875 [==============================] - 56s 30ms/step - loss: 0.1701 - accuracy: 0.9482 - val_loss: 0.0626 -
val_accuracy: 0.9813
Epoch 5/19
1875/1875 [==============================] - 67s 36ms/step - loss: 0.1538 - accuracy: 0.9538 - val_loss: 0.0530 -
val_accuracy: 0.9830
Epoch 6/19
1875/1875 [==============================] - 69s 37ms/step - loss: 0.1463 - accuracy: 0.9567 - val_loss: 0.2270 -
val_accuracy: 0.9480
Epoch 7/19
1875/1875 [==============================] - 65s 35ms/step - loss: 0.1457 - accuracy: 0.9570 - val_loss: 0.0617 -
val_accuracy: 0.9834
Epoch 8/19
1875/1875 [==============================] - 62s 33ms/step - loss: 0.1384 - accuracy: 0.9596 - val_loss: 0.0424 -
val_accuracy: 0.9884
Epoch 9/19
1875/1875 [==============================] - 60s 32ms/step - loss: 0.1311 - accuracy: 0.9609 - val_loss: 0.0472 -
val_accuracy: 0.9866
Epoch 10/19
1875/1875 [==============================] - 61s 32ms/step - loss: 0.1275 - accuracy: 0.9627 - val_loss: 0.0588 -
val_accuracy: 0.9849
Epoch 11/19
1875/1875 [==============================] - 59s 31ms/step - loss: 0.1234 - accuracy: 0.9642 - val_loss: 0.0581 -
val_accuracy: 0.9822
Epoch 12/19
1875/1875 [==============================] - 63s 33ms/step - loss: 0.1200 - accuracy: 0.9643 - val_loss: 0.0547 -
val_accuracy: 0.9840
Epoch 13/19
1875/1875 [==============================] - 62s 33ms/step - loss: 0.1180 - accuracy: 0.9651 - val_loss: 0.0587 -
val_accuracy: 0.9826
Epoch 14/19
1875/1875 [==============================] - 64s 34ms/step - loss: 0.1157 - accuracy: 0.9662 - val_loss: 0.0424 -
val_accuracy: 0.9870
Epoch 15/19
1875/1875 [==============================] - 57s 30ms/step - loss: 0.1162 - accuracy: 0.9664 - val_loss: 0.0426 -
val_accuracy: 0.9884
Epoch 16/19
1875/1875 [==============================] - 57s 30ms/step - loss: 0.1084 - accuracy: 0.9694 - val_loss: 0.0442 -
val_accuracy: 0.9864
Epoch 17/19
1875/1875 [==============================] - 56s 30ms/step - loss: 0.1080 - accuracy: 0.9681 - val_loss: 0.0404 -
val_accuracy: 0.9876
Epoch 18/19
1875/1875 [==============================] - 55s 29ms/step - loss: 0.1089 - accuracy: 0.9681 - val_loss: 0.0453 -
val_accuracy: 0.9876
Epoch 19/19
1875/1875 [==============================] - 58s 31ms/step - loss: 0.1017 - accuracy: 0.9702 - val_loss: 0.0387 -
val_accuracy: 0.9888
```

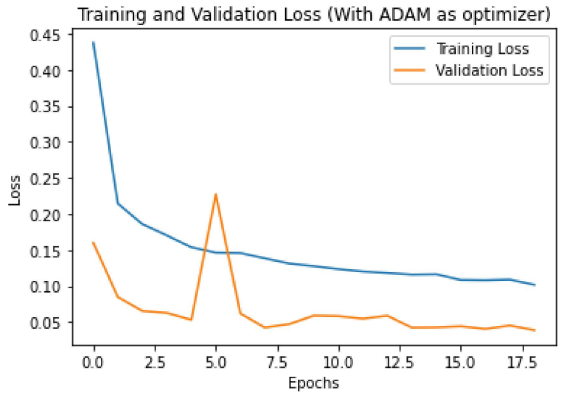
In [63]:

```
## checking for overfitting
# Plot training and validation accuracy

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history5.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history5.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With ADAM as optimizer)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [64]:

```
## checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history5.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history5.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss (With ADAM as optimizer)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [65]:

```
#make predictions
y_pred5 = np.argmax(cnn_model5.predict(x_test), axis=-1)
```

```
313/313 [==============================] - 3s 10ms/step
```

In [66]:

```
#Visualize the result
from sklearn.metrics import classification_report
cnn_classification_report5 = classification_report(y_test, y_pred5,output_dict=True)

print(classification_report(y_test, y_pred5))
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       980
           1       1.00      0.99      0.99      1135
           2       0.97      0.98      0.98      1032
           3       0.99      1.00      0.99      1010
           4       0.99      0.99      0.99       982
           5       0.98      0.98      0.98       892
           6       0.99      0.98      0.98       958
           7       0.98      0.99      0.99      1028
           8       1.00      1.00      1.00       974
           9       0.99      0.99      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

Observations:

- The model's accuracy decreased by 0.03% using Adam as the Optimizer while the validation loss increased by 3.47% with 0.0387 being the least validation loss for the Adam optimizer as against 0.0374 for the SGD optimizer

- SGD optimizer seems to perform better than Adam optimizer here

```
OPTIMIZER: RMSprop
The following parameters will be kept constant : Learning_rate = 0.001, Batch_size = 32 and Epochs = 19
```

In [69]:

```
#Using RMSProp as Optimizer
cnn_model6_1 = Sequential()

# Add the 1st convolutional block
cnn_model6_1.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),activation = 'relu',padding = 'same
cnn_model6_1.add(MaxPooling2D(pool_size=(2,2)))

# Add the 2nd convolutional block
cnn_model6_1.add(Conv2D(filters=64,kernel_size=(3,3),activation = 'relu'))
cnn_model6_1.add(MaxPooling2D(pool_size=(2,2)))

# Add the 3rd convolutional block
cnn_model6_1.add(Conv2D(filters=128,kernel_size=(3,3),activation = 'relu'))
cnn_model6_1.add(MaxPooling2D(pool_size=(2,2)))

# Flatten the output of the convolutional layers
cnn_model6_1.add(Flatten())
cnn_model6_1.add(BatchNormalization())

# Add a fully connected layer
cnn_model6_1.add(Dense(128,activation = 'relu'))
cnn_model6_1.add(Dropout(0.5))
cnn_model6_1.add(Dense(10,activation = 'softmax'))

#compiling the model
cnn_model6_1.compile(optimizer=RMSprop(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accura
```

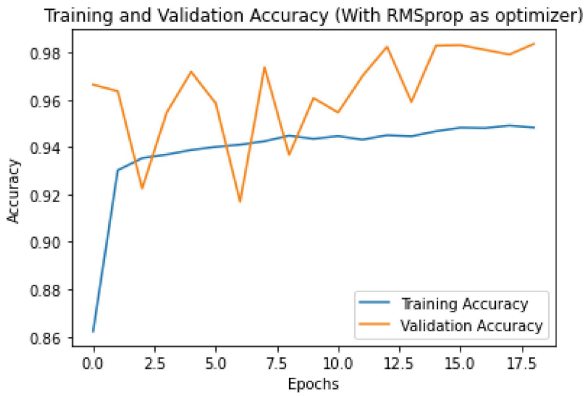
In [70]:

```
#Training and evaluating the model
history6_1 = cnn_model6_1.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_si
        epochs=19, validation_data=(x_test, keras.utils.to_categorical(y_test)))
```

```
Epoch 1/19
1875/1875 [==============================] - 57s 30ms/step - loss: 0.4355 - accuracy: 0.8623 - val_loss: 0.1360 -
val_accuracy: 0.9664
Epoch 2/19
1875/1875 [==============================] - 57s 30ms/step - loss: 0.2464 - accuracy: 0.9302 - val_loss: 0.1368 -
val_accuracy: 0.9636
Epoch 3/19
1875/1875 [==============================] - 52s 28ms/step - loss: 0.2378 - accuracy: 0.9354 - val_loss: 0.7461 -
val_accuracy: 0.9225
Epoch 4/19
1875/1875 [==============================] - 54s 29ms/step - loss: 0.2436 - accuracy: 0.9368 - val_loss: 0.3176 -
val_accuracy: 0.9546
Epoch 5/19
1875/1875 [==============================] - 55s 29ms/step - loss: 0.2393 - accuracy: 0.9388 - val_loss: 0.2262 -
val_accuracy: 0.9718
Epoch 6/19
1875/1875 [==============================] - 53s 28ms/step - loss: 0.2381 - accuracy: 0.9401 - val_loss: 0.3486 -
val_accuracy: 0.9586
Epoch 7/19
1875/1875 [==============================] - 52s 28ms/step - loss: 0.2416 - accuracy: 0.9410 - val_loss: 0.6672 -
val_accuracy: 0.9170
Epoch 8/19
1875/1875 [==============================] - 53s 28ms/step - loss: 0.2382 - accuracy: 0.9425 - val_loss: 0.1558 -
val_accuracy: 0.9737
Epoch 9/19
1875/1875 [==============================] - 53s 28ms/step - loss: 0.2384 - accuracy: 0.9449 - val_loss: 0.5048 -
val_accuracy: 0.9367
Epoch 10/19
1875/1875 [==============================] - 52s 27ms/step - loss: 0.2370 - accuracy: 0.9435 - val_loss: 0.2993 -
val_accuracy: 0.9607
Epoch 11/19
1875/1875 [==============================] - 56s 30ms/step - loss: 0.2338 - accuracy: 0.9446 - val_loss: 0.3154 -
val_accuracy: 0.9546
Epoch 12/19
1875/1875 [==============================] - 58s 31ms/step - loss: 0.2330 - accuracy: 0.9431 - val_loss: 0.2216 -
val_accuracy: 0.9700
Epoch 13/19
1875/1875 [==============================] - 56s 30ms/step - loss: 0.2309 - accuracy: 0.9451 - val_loss: 0.0969 -
val_accuracy: 0.9824
Epoch 14/19
1875/1875 [==============================] - 56s 30ms/step - loss: 0.2270 - accuracy: 0.9446 - val_loss: 0.2766 -
val_accuracy: 0.9591
Epoch 15/19
1875/1875 [==============================] - 56s 30ms/step - loss: 0.2250 - accuracy: 0.9467 - val_loss: 0.1011 -
val_accuracy: 0.9829
Epoch 16/19
1875/1875 [==============================] - 55s 29ms/step - loss: 0.2196 - accuracy: 0.9482 - val_loss: 0.1076 -
val_accuracy: 0.9831
Epoch 17/19
1875/1875 [==============================] - 54s 29ms/step - loss: 0.2155 - accuracy: 0.9480 - val_loss: 0.1352 -
val_accuracy: 0.9811
Epoch 18/19
1875/1875 [==============================] - 54s 29ms/step - loss: 0.2093 - accuracy: 0.9492 - val_loss: 0.1090 -
val_accuracy: 0.9791
Epoch 19/19
1875/1875 [==============================] - 56s 30ms/step - loss: 0.2134 - accuracy: 0.9483 - val_loss: 0.1013 -
val_accuracy: 0.9836
```

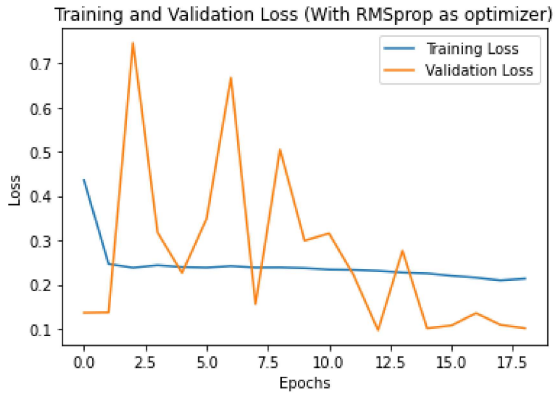
In [71]:

```
# checking for overfitting

# Plot training and validation accuracy

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history6_1.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history6_1.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With RMSprop as optimizer)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [72]:

```
## checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history6_1.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history6_1.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss (With RMSprop as optimizer)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [73]:

```
#make predictions
y_pred6_1 = np.argmax(cnn_model6_1.predict(x_test), axis=-1)
```

```
313/313 [==============================] - 3s 8ms/step
```

In [74]:

```
#Visualize the result
from sklearn.metrics import classification_report
cnn_classification_report6_1 = classification_report(y_test, y_pred6_1,output_dict=True)

print(classification_report(y_test, y_pred6_1))
```

```
              precision    recall  f1-score   support

           0       1.00      0.97      0.99       980
           1       1.00      0.99      0.99      1135
           2       0.95      0.99      0.97      1032
           3       0.98      1.00      0.99      1010
           4       0.99      0.98      0.99       982
           5       0.99      0.97      0.98       892
           6       0.98      0.97      0.97       958
           7       0.98      0.98      0.98      1028
           8       0.99      1.00      0.99       974
           9       0.99      0.99      0.99      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

In [75]:
```
df_report6_1 = pd.DataFrame(cnn_classification_report6_1).transpose()
df_report6_1
```

Out[75]:

	precision	recall	f1-score	support
0	1.000000	0.972449	0.986032	980.0000
1	0.997331	0.987665	0.992475	1135.0000
2	0.947075	0.988372	0.967283	1032.0000
3	0.980564	0.999010	0.989701	1010.0000
4	0.994845	0.982688	0.988730	982.0000
5	0.988623	0.974215	0.981366	892.0000
6	0.976915	0.971816	0.974359	958.0000
7	0.982370	0.975681	0.979014	1028.0000
8	0.985772	0.995893	0.990807	974.0000
9	0.985149	0.986125	0.985636	1009.0000
accuracy	0.983600	0.983600	0.983600	0.9836
macro avg	0.983864	0.983392	0.983540	10000.0000
weighted avg	0.983843	0.983600	0.983633	10000.0000

Observations:

- The model's accuracy decreased by 0.56% using RMSprop as the Optimizer while the validation loss increased by 159.09% with 0.0969 being the least validation loss for the RMSprop optimizer as against 0.0374 for the SGD optimizer.
- SGD optimizer performed better than RMSprop

Question(b)

Report how changes to the number of convolution blocks affect the performance of your model quantitatively?

Increasing the convolution blocks from 3 to 4 with learning rate =0.001, batchsize = 32, epoch = 19 and optimizer = SGD, regularizer = batch normalization

In [76]:
```
from keras.layers import ZeroPadding2D
```

In [82]:

```
# Define the model architecture
cnn_model7_3 = Sequential()

# Add the 1st convolutional block
cnn_model7_3.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),activation = 'relu',padding = 'same
cnn_model7_3.add(BatchNormalization())
cnn_model7_3.add(MaxPooling2D(pool_size=(2,2)))

# Add the 2nd convolutional block
cnn_model7_3.add(Conv2D(filters=64,kernel_size=(3,3),activation = 'relu'))
cnn_model7_3.add(BatchNormalization())
cnn_model7_3.add(MaxPooling2D(pool_size=(2,2)))

# Add the 3rd convolutional block
cnn_model7_3.add(Conv2D(filters=128,kernel_size=(3,3),activation = 'relu'))
cnn_model7_3.add(BatchNormalization())
cnn_model7_3.add(MaxPooling2D(pool_size=(2,2)))
cnn_model7_3.add(ZeroPadding2D(padding=((1, 1), (1, 1)))) # Add padding layer to resolve Negative dimension s

# Add the 4th convolutional block
cnn_model7_3.add(Conv2D(filters=256,kernel_size=(3,3),activation = 'relu'))
cnn_model7_3.add(BatchNormalization())
cnn_model7_3.add(MaxPooling2D(pool_size=(2,2)))

# Flatten the output of the convolutional layers
cnn_model7_3.add(Flatten())
cnn_model7_3.add(BatchNormalization())
cnn_model7_3.add(Dense(128,activation = 'relu'))
cnn_model7_3.add(Dropout(0.5))
cnn_model7_3.add(Dense(10,activation = 'softmax'))
#compiling the model
cnn_model7_3.compile(optimizer=SGD(learning_rate=0.001, momentum=0.9), loss='categorical_crossentropy', metri
```

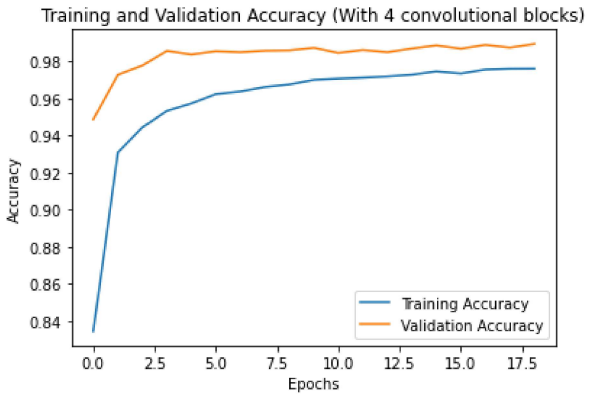
In [83]:

```
#Training and evaluating the model
history7_3 = cnn_model7_3.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_si
        epochs=19, validation_data=(x_test, keras.utils.to_categorical(y_test)))
```

```
Epoch 1/19
1875/1875 [==============================] - 112s 59ms/step - loss: 0.5234 - accuracy: 0.8346 - val_loss: 0.1548
- val_accuracy: 0.9487
Epoch 2/19
1875/1875 [==============================] - 108s 57ms/step - loss: 0.2286 - accuracy: 0.9308 - val_loss: 0.0824
- val_accuracy: 0.9727
Epoch 3/19
1875/1875 [==============================] - 107s 57ms/step - loss: 0.1839 - accuracy: 0.9444 - val_loss: 0.0638
- val_accuracy: 0.9777
Epoch 4/19
1875/1875 [==============================] - 108s 58ms/step - loss: 0.1540 - accuracy: 0.9532 - val_loss: 0.0452
- val_accuracy: 0.9856
Epoch 5/19
1875/1875 [==============================] - 111s 59ms/step - loss: 0.1409 - accuracy: 0.9573 - val_loss: 0.0519
- val_accuracy: 0.9837
Epoch 6/19
1875/1875 [==============================] - 105s 56ms/step - loss: 0.1254 - accuracy: 0.9623 - val_loss: 0.0413
- val_accuracy: 0.9854
Epoch 7/19
1875/1875 [==============================] - 104s 56ms/step - loss: 0.1200 - accuracy: 0.9637 - val_loss: 0.0470
- val_accuracy: 0.9849
Epoch 8/19
1875/1875 [==============================] - 106s 56ms/step - loss: 0.1092 - accuracy: 0.9662 - val_loss: 0.0423
- val_accuracy: 0.9857
Epoch 9/19
1875/1875 [==============================] - 108s 58ms/step - loss: 0.1060 - accuracy: 0.9675 - val_loss: 0.0443
- val_accuracy: 0.9859
Epoch 10/19
1875/1875 [==============================] - 109s 58ms/step - loss: 0.0981 - accuracy: 0.9700 - val_loss: 0.0419
- val_accuracy: 0.9873
Epoch 11/19
1875/1875 [==============================] - 112s 60ms/step - loss: 0.0978 - accuracy: 0.9707 - val_loss: 0.0465
- val_accuracy: 0.9845
Epoch 12/19
1875/1875 [==============================] - 99s 53ms/step - loss: 0.0945 - accuracy: 0.9712 - val_loss: 0.0434 -
val_accuracy: 0.9861
Epoch 13/19
1875/1875 [==============================] - 97s 52ms/step - loss: 0.0920 - accuracy: 0.9718 - val_loss: 0.0465 -
val_accuracy: 0.9849
Epoch 14/19
1875/1875 [==============================] - 96s 51ms/step - loss: 0.0904 - accuracy: 0.9728 - val_loss: 0.0426 -
val_accuracy: 0.9869
Epoch 15/19
1875/1875 [==============================] - 95s 51ms/step - loss: 0.0856 - accuracy: 0.9745 - val_loss: 0.0418 -
val_accuracy: 0.9886
Epoch 16/19
1875/1875 [==============================] - 93s 50ms/step - loss: 0.0868 - accuracy: 0.9734 - val_loss: 0.0425 -
val_accuracy: 0.9868
Epoch 17/19
1875/1875 [==============================] - 110s 59ms/step - loss: 0.0813 - accuracy: 0.9755 - val_loss: 0.0360
- val_accuracy: 0.9889
Epoch 18/19
1875/1875 [==============================] - 108s 58ms/step - loss: 0.0797 - accuracy: 0.9760 - val_loss: 0.0391
- val_accuracy: 0.9874
Epoch 19/19
1875/1875 [==============================] - 104s 56ms/step - loss: 0.0798 - accuracy: 0.9760 - val_loss: 0.0357
- val_accuracy: 0.9894
```

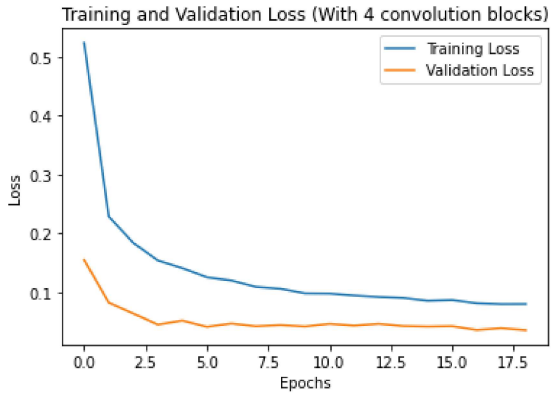
In [84]:

```
## checking for overfitting

# Plot training and validation accuracy

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history7_3.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history7_3.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With 4 convolutional blocks)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [85]:

```
## checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history7_3.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history7_3.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss (With 4 convolution blocks)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [86]:

```
#make predictions
y_pred7_1 = np.argmax(cnn_model7_3.predict(x_test), axis=-1)
```

```
313/313 [==============================] - 4s 12ms/step
```

In [87]:

```
#Visualize the result
from sklearn.metrics import classification_report
cnn_classification_report7_1 = classification_report(y_test, y_pred7_1,output_dict=True)

print(classification_report(y_test, y_pred7_1))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       980
           1       1.00      0.99      1.00      1135
           2       0.98      0.98      0.98      1032
           3       0.98      1.00      0.99      1010
           4       0.99      0.99      0.99       982
           5       0.98      0.98      0.98       892
           6       0.99      0.98      0.98       958
           7       0.99      1.00      0.99      1028
           8       0.99      0.99      0.99       974
           9       1.00      0.98      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

In [89]:

```
df_report7_1 = pd.DataFrame(cnn_classification_report7_1).transpose()
df_report7_1
```

Out[89]:

	precision	recall	f1-score	support
0	0.995923	0.996939	0.996430	980.0000
1	0.998232	0.994714	0.996470	1135.0000
2	0.975822	0.977713	0.976767	1032.0000
3	0.984390	0.999010	0.991646	1010.0000
4	0.986829	0.991853	0.989335	982.0000
5	0.982022	0.979821	0.980920	892.0000
6	0.986359	0.981211	0.983778	958.0000
7	0.991288	0.996109	0.993692	1028.0000
8	0.993846	0.994867	0.994356	974.0000
9	0.997982	0.980178	0.989000	1009.0000
accuracy	0.989400	0.989400	0.989400	0.9894
macro avg	0.989269	0.989241	0.989239	10000.0000
weighted avg	0.989425	0.989400	0.989397	10000.0000

Increasing the convolution blocks from 4 to 5 with learning rate =0.001, batchsize = 32, epoch = 19 and optimizer = SGD, regularizer = batch normalization

In [96]:

```
# Define the model architecture
cnn_model8_1 = Sequential()

# Add the 1st convolutional block
cnn_model8_1.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),activation = 'relu',padding = 'sam
cnn_model8_1.add(BatchNormalization())
cnn_model8_1.add(MaxPooling2D(pool_size=(2,2)))

# Add the 2nd convolutional block
cnn_model8_1.add(Conv2D(filters=64,kernel_size=(3,3),activation = 'relu',padding = 'same'))
cnn_model8_1.add(BatchNormalization())
cnn_model8_1.add(MaxPooling2D(pool_size=(2,2)))

# Add the 3rd convolutional block
cnn_model8_1.add(Conv2D(filters=128,kernel_size=(3,3),activation = 'relu',padding = 'same'))
cnn_model8_1.add(BatchNormalization())
cnn_model8_1.add(MaxPooling2D(pool_size=(2,2)))


# Add the 4th convolutional block
cnn_model8_1.add(Conv2D(filters=256,kernel_size=(3,3),activation = 'relu',padding = 'same'))
cnn_model8_1.add(BatchNormalization())
cnn_model8_1.add(MaxPooling2D(pool_size=(2,2)))
cnn_model8_1.add(ZeroPadding2D(padding=((1, 1), (1, 1)))) # Add padding layer to resolve Negative dimension s

# Add the 5th convolutional block
cnn_model8_1.add(Conv2D(filters=512,kernel_size=(3,3),activation = 'relu',padding = 'same'))
cnn_model8_1.add(BatchNormalization())
cnn_model8_1.add(MaxPooling2D(pool_size=(2,2)))

# Flatten the output of the convolutional layers
cnn_model8_1.add(Flatten())
cnn_model8_1.add(BatchNormalization())
cnn_model8_1.add(Dense(128,activation = 'relu'))
cnn_model8_1.add(Dropout(0.5))
cnn_model8_1.add(Dense(10,activation = 'softmax'))
#compiling the model
cnn_model8_1.compile(optimizer=SGD(learning_rate=0.001, momentum=0.9), loss='categorical_crossentropy', metri
```

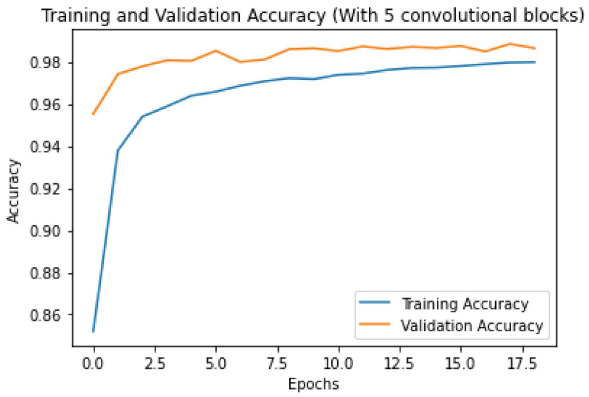
In [97]:

```
#Training and evaluating the model
history8_1 = cnn_model8_1.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_si
        epochs=19, validation_data=(x_test, keras.utils.to_categorical(y_test)))
```

```
Epoch 1/19
1875/1875 [==============================] - 376s 199ms/step - loss: 0.4722 - accuracy: 0.8521 - val_loss: 0.1398
- val_accuracy: 0.9552
Epoch 2/19
1875/1875 [==============================] - 420s 224ms/step - loss: 0.2053 - accuracy: 0.9378 - val_loss: 0.0844
- val_accuracy: 0.9741
Epoch 3/19
1875/1875 [==============================] - 417s 222ms/step - loss: 0.1548 - accuracy: 0.9539 - val_loss: 0.0701
- val_accuracy: 0.9778
Epoch 4/19
1875/1875 [==============================] - 346s 184ms/step - loss: 0.1372 - accuracy: 0.9588 - val_loss: 0.0584
- val_accuracy: 0.9807
Epoch 5/19
1875/1875 [==============================] - 328s 175ms/step - loss: 0.1207 - accuracy: 0.9639 - val_loss: 0.0592
- val_accuracy: 0.9804
Epoch 6/19
1875/1875 [==============================] - 396s 211ms/step - loss: 0.1097 - accuracy: 0.9658 - val_loss: 0.0430
- val_accuracy: 0.9852
Epoch 7/19
1875/1875 [==============================] - 466s 248ms/step - loss: 0.1023 - accuracy: 0.9686 - val_loss: 0.0648
- val_accuracy: 0.9799
Epoch 8/19
1875/1875 [==============================] - 437s 233ms/step - loss: 0.0962 - accuracy: 0.9708 - val_loss: 0.0582
- val_accuracy: 0.9811
Epoch 9/19
1875/1875 [==============================] - 449s 240ms/step - loss: 0.0919 - accuracy: 0.9722 - val_loss: 0.0433
- val_accuracy: 0.9860
Epoch 10/19
1875/1875 [==============================] - 368s 196ms/step - loss: 0.0934 - accuracy: 0.9718 - val_loss: 0.0419
- val_accuracy: 0.9864
Epoch 11/19
1875/1875 [==============================] - 385s 205ms/step - loss: 0.0874 - accuracy: 0.9738 - val_loss: 0.0445
- val_accuracy: 0.9851
Epoch 12/19
1875/1875 [==============================] - 399s 213ms/step - loss: 0.0844 - accuracy: 0.9744 - val_loss: 0.0386
- val_accuracy: 0.9873
Epoch 13/19
1875/1875 [==============================] - 387s 206ms/step - loss: 0.0793 - accuracy: 0.9761 - val_loss: 0.0422
- val_accuracy: 0.9861
Epoch 14/19
1875/1875 [==============================] - 400s 213ms/step - loss: 0.0770 - accuracy: 0.9771 - val_loss: 0.0391
- val_accuracy: 0.9871
Epoch 15/19
1875/1875 [==============================] - 460s 245ms/step - loss: 0.0764 - accuracy: 0.9772 - val_loss: 0.0400
- val_accuracy: 0.9866
Epoch 16/19
1875/1875 [==============================] - 592s 316ms/step - loss: 0.0734 - accuracy: 0.9780 - val_loss: 0.0372
- val_accuracy: 0.9875
Epoch 17/19
1875/1875 [==============================] - 440s 234ms/step - loss: 0.0695 - accuracy: 0.9789 - val_loss: 0.0473
- val_accuracy: 0.9849
Epoch 18/19
1875/1875 [==============================] - 438s 233ms/step - loss: 0.0684 - accuracy: 0.9797 - val_loss: 0.0362
- val_accuracy: 0.9885
Epoch 19/19
1875/1875 [==============================] - 437s 233ms/step - loss: 0.0673 - accuracy: 0.9798 - val_loss: 0.0400
- val_accuracy: 0.9865
```

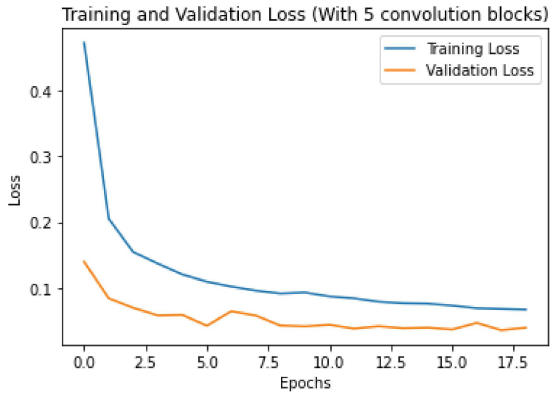
In [98]:

```
## checking for overfitting

# Plot training and validation accuracy

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history8_1.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history8_1.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With 5 convolutional blocks)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [99]:

```
## checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history8_1.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history8_1.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss (With 5 convolution blocks)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



Observations:

- The model's accuracy decreased by 0.03% when the number of convolution blocks was increased to 4 from 3 while the validation loss decreased by 4.53%.
- Increasing the number of convolution blocks from 4 to 5 showed a decrease in the accuracy by 0.26% from an accuracy of 0.989100 at 3 convolution blocks to 0.9865 at 5 blocks while the validation loss increased by 7.00% from 0.0357 at 3 blocks to 0.0400 at 5 blocks.
- However, considering the time it took to train the model and the weighted average accuracy which is higher in the model with three (3) convolution blocks, the model with 3 convolution blocks is a more optimal option

Question(c)

What is the effect of varying learning rates on the performance of the CNN algorithm?

Optimizer = SDG, Learning Rate = 0.01, epoch = 19, and batch size = 32 (from my constant)

In [100]:

```
cnn_model9 = Sequential()

cnn_model9.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),padding = 'same'))
cnn_model9.add(BatchNormalization())
cnn_model9.add(Activation('relu'))
cnn_model9.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9.add(Conv2D(filters=64,kernel_size=(3,3)))
cnn_model9.add(BatchNormalization())
cnn_model9.add(Activation('relu'))
cnn_model9.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9.add(Conv2D(filters=128,kernel_size=(3,3)))
cnn_model9.add(BatchNormalization())
cnn_model9.add(Activation('relu'))
cnn_model9.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9.add(Flatten())
cnn_model9.add(BatchNormalization())
cnn_model9.add(Dense(128,activation = 'relu'))
cnn_model9.add(Dropout(0.5))
cnn_model9.add(Dense(10,activation = 'softmax'))

#compiling the model
sgd = SGD(learning_rate=0.01, momentum=0.9)
cnn_model9.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
```

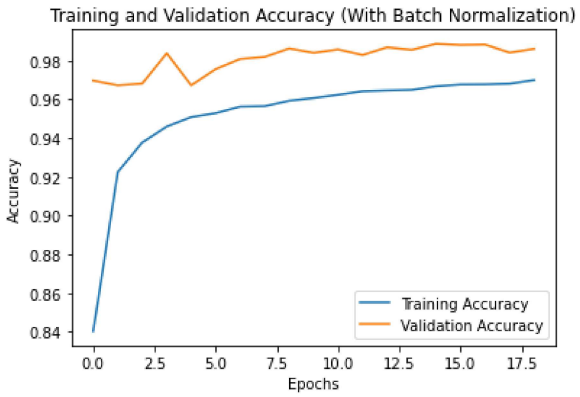
In [101]:

```
#Training and evaluating the model
history9 = cnn_model9.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_size=3
         epochs = 19, validation_data=(x_test, keras.utils.to_categorical(y_test)))
```

```
Epoch 1/19
1875/1875 [==============================] - 115s 60ms/step - loss: 0.5009 - accuracy: 0.8404 - val_loss: 0.0961
- val_accuracy: 0.9696
Epoch 2/19
1875/1875 [==============================] - 119s 64ms/step - loss: 0.2556 - accuracy: 0.9225 - val_loss: 0.1078
- val_accuracy: 0.9672
Epoch 3/19
1875/1875 [==============================] - 148s 78ms/step - loss: 0.2097 - accuracy: 0.9377 - val_loss: 0.1033
- val_accuracy: 0.9681
Epoch 4/19
1875/1875 [==============================] - 127s 68ms/step - loss: 0.1780 - accuracy: 0.9459 - val_loss: 0.0568
- val_accuracy: 0.9838
Epoch 5/19
1875/1875 [==============================] - 113s 60ms/step - loss: 0.1669 - accuracy: 0.9508 - val_loss: 0.1134
- val_accuracy: 0.9672
Epoch 6/19
1875/1875 [==============================] - 118s 63ms/step - loss: 0.1590 - accuracy: 0.9529 - val_loss: 0.0840
- val_accuracy: 0.9756
Epoch 7/19
1875/1875 [==============================] - 103s 55ms/step - loss: 0.1451 - accuracy: 0.9562 - val_loss: 0.0600
- val_accuracy: 0.9808
Epoch 8/19
1875/1875 [==============================] - 122s 65ms/step - loss: 0.1413 - accuracy: 0.9565 - val_loss: 0.0630
- val_accuracy: 0.9819
Epoch 9/19
1875/1875 [==============================] - 118s 63ms/step - loss: 0.1356 - accuracy: 0.9592 - val_loss: 0.0444
- val_accuracy: 0.9862
Epoch 10/19
1875/1875 [==============================] - 111s 59ms/step - loss: 0.1292 - accuracy: 0.9606 - val_loss: 0.0518
- val_accuracy: 0.9840
Epoch 11/19
1875/1875 [==============================] - 111s 59ms/step - loss: 0.1271 - accuracy: 0.9623 - val_loss: 0.0441
- val_accuracy: 0.9857
Epoch 12/19
1875/1875 [==============================] - 110s 58ms/step - loss: 0.1227 - accuracy: 0.9641 - val_loss: 0.0599
- val_accuracy: 0.9829
Epoch 13/19
1875/1875 [==============================] - 102s 54ms/step - loss: 0.1199 - accuracy: 0.9645 - val_loss: 0.0465
- val_accuracy: 0.9868
Epoch 14/19
1875/1875 [==============================] - 107s 57ms/step - loss: 0.1171 - accuracy: 0.9648 - val_loss: 0.0489
- val_accuracy: 0.9855
Epoch 15/19
1875/1875 [==============================] - 108s 58ms/step - loss: 0.1117 - accuracy: 0.9667 - val_loss: 0.0383
- val_accuracy: 0.9886
Epoch 16/19
1875/1875 [==============================] - 105s 56ms/step - loss: 0.1103 - accuracy: 0.9676 - val_loss: 0.0381
- val_accuracy: 0.9880
Epoch 17/19
1875/1875 [==============================] - 112s 60ms/step - loss: 0.1079 - accuracy: 0.9677 - val_loss: 0.0391
- val_accuracy: 0.9882
Epoch 18/19
1875/1875 [==============================] - 116s 62ms/step - loss: 0.1077 - accuracy: 0.9680 - val_loss: 0.0555
- val_accuracy: 0.9841
Epoch 19/19
1875/1875 [==============================] - 103s 55ms/step - loss: 0.1011 - accuracy: 0.9699 - val_loss: 0.0482
- val_accuracy: 0.9860
```

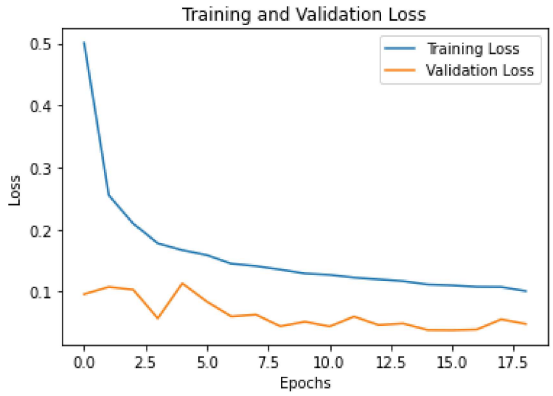
In [102]:

```
## checking for overfitting
'''
# Plot training and validation accuracy

'''

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history9.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history9.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With Batch Normalization)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [138]:

```
## checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history9.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history9.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [104]:

```
#make predictions
y_pred9 = np.argmax(cnn_model9.predict(x_test), axis=-1)
```

```
313/313 [==============================] - 5s 15ms/step
```

In [105]:

```
#Visualize the result
from sklearn.metrics import classification_report
cnn_classification_report9_1 = classification_report(y_test, y_pred9,output_dict=True)

print(classification_report(y_test, y_pred9))
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       980
           1       1.00      0.99      0.99      1135
           2       0.97      0.97      0.97      1032
           3       0.99      1.00      0.99      1010
           4       1.00      0.97      0.98       982
           5       0.99      0.98      0.98       892
           6       0.98      0.98      0.98       958
           7       0.98      1.00      0.99      1028
           8       1.00      0.99      1.00       974
           9       0.97      1.00      0.98      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

In [106]:

```
df_report9_1 = pd.DataFrame(cnn_classification_report9_1).transpose()
df_report9_1
```

Out[106]:

	precision	recall	f1-score	support
0	0.992872	0.994898	0.993884	980.000
1	0.997329	0.986784	0.992028	1135.000
2	0.970106	0.974806	0.972450	1032.000
3	0.990186	0.999010	0.994579	1010.000
4	0.997895	0.965377	0.981366	982.000
5	0.985277	0.975336	0.980282	892.000
6	0.980084	0.975992	0.978033	958.000
7	0.983654	0.995136	0.989362	1028.000
8	0.997938	0.993840	0.995885	974.000
9	0.965451	0.997027	0.980985	1009.000
accuracy	0.986000	0.986000	0.986000	0.986
macro avg	0.986079	0.985821	0.985885	10000.000
weighted avg	0.986127	0.986000	0.985999	10000.000

Observations:

- The model's accuracy decreased by 28.73% from 0.989100 to 0.986000 when the learning rate was changed to 0.01 from 0.001 while the validation loss increased by 29.19% from 0.0374 to 0.0482.

Optimizer = SDG, Learning Rate = 0.005, epoch = 19, and batch size = 32 (from my constant)

In [107]:

```
cnn_model9_1 = Sequential()

cnn_model9_1.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),padding = 'same'))
cnn_model9_1.add(BatchNormalization())
cnn_model9_1.add(Activation('relu'))
cnn_model9_1.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9_1.add(Conv2D(filters=64,kernel_size=(3,3)))
cnn_model9_1.add(BatchNormalization())
cnn_model9_1.add(Activation('relu'))
cnn_model9_1.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9_1.add(Conv2D(filters=128,kernel_size=(3,3)))
cnn_model9_1.add(BatchNormalization())
cnn_model9_1.add(Activation('relu'))
cnn_model9_1.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9_1.add(Flatten())
cnn_model9_1.add(BatchNormalization())
cnn_model9_1.add(Dense(128,activation = 'relu'))
cnn_model9_1.add(Dropout(0.5))
cnn_model9_1.add(Dense(10,activation = 'softmax'))

#compiling the model
sgd = SGD(learning_rate=0.005, momentum=0.9)
cnn_model9_1.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
```

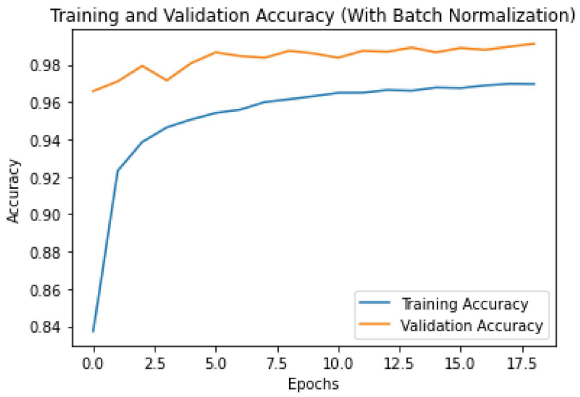
In [108]:

```
#Training and evaluating the model
history9_1 = cnn_model9_1.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_si
        epochs = 19, validation_data=(x_test, keras.utils.to_categorical(y_test)))
```

```
Epoch 1/19
1875/1875 [==============================] - 112s 59ms/step - loss: 0.5071 - accuracy: 0.8374 - val_loss: 0.1041
- val_accuracy: 0.9659
Epoch 2/19
1875/1875 [==============================] - 109s 58ms/step - loss: 0.2480 - accuracy: 0.9233 - val_loss: 0.0933
- val_accuracy: 0.9711
Epoch 3/19
1875/1875 [==============================] - 120s 64ms/step - loss: 0.2007 - accuracy: 0.9386 - val_loss: 0.0654
- val_accuracy: 0.9794
Epoch 4/19
1875/1875 [==============================] - 113s 60ms/step - loss: 0.1741 - accuracy: 0.9465 - val_loss: 0.0923
- val_accuracy: 0.9716
Epoch 5/19
1875/1875 [==============================] - 103s 55ms/step - loss: 0.1604 - accuracy: 0.9507 - val_loss: 0.0619
- val_accuracy: 0.9809
Epoch 6/19
1875/1875 [==============================] - 109s 58ms/step - loss: 0.1487 - accuracy: 0.9542 - val_loss: 0.0450
- val_accuracy: 0.9867
Epoch 7/19
1875/1875 [==============================] - 112s 60ms/step - loss: 0.1430 - accuracy: 0.9560 - val_loss: 0.0490
- val_accuracy: 0.9847
Epoch 8/19
1875/1875 [==============================] - 112s 60ms/step - loss: 0.1321 - accuracy: 0.9601 - val_loss: 0.0553
- val_accuracy: 0.9838
Epoch 9/19
1875/1875 [==============================] - 115s 61ms/step - loss: 0.1153 - accuracy: 0.9649 - val_loss: 0.0539
- val_accuracy: 0.9838
Epoch 12/19
1875/1875 [==============================] - 110s 59ms/step - loss: 0.1161 - accuracy: 0.9650 - val_loss: 0.0406
- val_accuracy: 0.9875
Epoch 13/19
1875/1875 [==============================] - 117s 62ms/step - loss: 0.1132 - accuracy: 0.9661 - val_loss: 0.0392
- val_accuracy: 0.9892
Epoch 15/19
1875/1875 [==============================] - 111s 59ms/step - loss: 0.1079 - accuracy: 0.9678 - val_loss: 0.0454
- val_accuracy: 0.9867
Epoch 16/19
1875/1875 [==============================] - 115s 61ms/step - loss: 0.1078 - accuracy: 0.9675 - val_loss: 0.0364
- val_accuracy: 0.9890
Epoch 17/19
1875/1875 [==============================] - 115s 61ms/step - loss: 0.1019 - accuracy: 0.9689 - val_loss: 0.0396
- val_accuracy: 0.9880
Epoch 18/19
1875/1875 [==============================] - 108s 58ms/step - loss: 0.0999 - accuracy: 0.9698 - val_loss: 0.0341
- val_accuracy: 0.9897
Epoch 19/19
1875/1875 [==============================] - 114s 61ms/step - loss: 0.1010 - accuracy: 0.9696 - val_loss: 0.0319
- val_accuracy: 0.9912
```

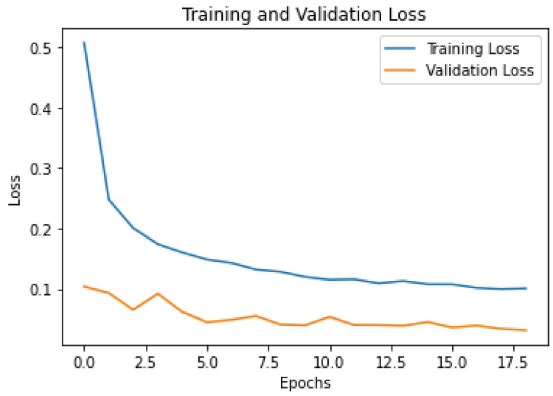
In [109]:

```
## checking for overfitting
'''
# Plot training and validation accuracy

'''

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history9_1.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history9_1.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With Batch Normalization)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [137]:

```
## checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history9_1.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history9_1.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [111]:

```
#make predictions
y_pred9_1 = np.argmax(cnn_model9_1.predict(x_test), axis=-1)
```

```
313/313 [==============================] - 5s 14ms/step
```

In [112]:

```
#Visualize the result
from sklearn.metrics import classification_report
cnn_classification_report9_2 = classification_report(y_test, y_pred9_1,output_dict=True)

print(classification_report(y_test, y_pred9_1))
```

```
              precision    recall  f1-score   support

           0       1.00      0.99      1.00       980
           1       1.00      0.99      1.00      1135
           2       0.98      0.98      0.98      1032
           3       0.99      1.00      1.00      1010
           4       1.00      1.00      1.00       982
           5       0.98      0.98      0.98       892
           6       0.99      0.98      0.99       958
           7       0.99      0.99      0.99      1028
           8       0.99      1.00      0.99       974
           9       1.00      0.99      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

In [113]:
```
df_report9_2 = pd.DataFrame(cnn_classification_report9_2).transpose()
df_report9_2
```

Out[113]:

	precision	recall	f1-score	support
0	0.997951	0.993878	0.995910	980.0000
1	0.998230	0.993833	0.996026	1135.0000
2	0.976923	0.984496	0.980695	1032.0000
3	0.993097	0.997030	0.995059	1010.0000
4	0.995931	0.996945	0.996438	982.0000
5	0.984287	0.983184	0.983735	892.0000
6	0.991570	0.982255	0.986890	958.0000
7	0.988383	0.993191	0.990781	1028.0000
8	0.988821	0.998973	0.993871	974.0000
9	0.996000	0.987116	0.991538	1009.0000
accuracy	0.991200	0.991200	0.991200	0.9912
macro avg	0.991119	0.991090	0.991094	10000.0000
weighted avg	0.991224	0.991200	0.991202	10000.0000

Observations:

- The model's accuracy increased by 0.21% from 0.989100 to 0.991200 when the learning rate was changed to 0.005 from 0.001 while the validation loss decreased by 14.77% from 0.0374 to 0.0319.

Optimizer = SDG, Learning Rate = 0.0025, epoch = 19, and batch size = 32 (from my constant)

In [114]:
```
cnn_model9_2 = Sequential()

cnn_model9_2.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),padding = 'same'))
cnn_model9_2.add(BatchNormalization())
cnn_model9_2.add(Activation('relu'))
cnn_model9_2.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9_2.add(Conv2D(filters=64,kernel_size=(3,3)))
cnn_model9_2.add(BatchNormalization())
cnn_model9_2.add(Activation('relu'))
cnn_model9_2.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9_2.add(Conv2D(filters=128,kernel_size=(3,3)))
cnn_model9_2.add(BatchNormalization())
cnn_model9_2.add(Activation('relu'))
cnn_model9_2.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9_2.add(Flatten())
cnn_model9_2.add(BatchNormalization())
cnn_model9_2.add(Dense(128,activation = 'relu'))
cnn_model9_2.add(Dropout(0.5))
cnn_model9_2.add(Dense(10,activation = 'softmax'))

#compiling the model
sgd = SGD(learning_rate=0.0025, momentum=0.9)
cnn_model9_2.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
```

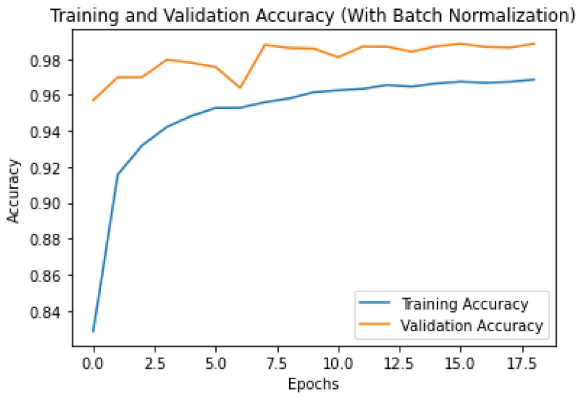
In [115]:

```
#Training and evaluating the model
history9_2 = cnn_model9_2.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_si
        epochs = 19, validation_data=(x_test, keras.utils.to_categorical(y_test)))
```

```
Epoch 1/19
1875/1875 [==============================] - 106s 56ms/step - loss: 0.5402 - accuracy: 0.8285 - val_loss: 0.1404
- val_accuracy: 0.9571
Epoch 2/19
1875/1875 [==============================] - 114s 61ms/step - loss: 0.2712 - accuracy: 0.9157 - val_loss: 0.0964
- val_accuracy: 0.9699
Epoch 3/19
1875/1875 [==============================] - 111s 59ms/step - loss: 0.2195 - accuracy: 0.9320 - val_loss: 0.0974
- val_accuracy: 0.9700
Epoch 4/19
1875/1875 [==============================] - 115s 61ms/step - loss: 0.1895 - accuracy: 0.9422 - val_loss: 0.0653
- val_accuracy: 0.9797
Epoch 5/19
1875/1875 [==============================] - 110s 59ms/step - loss: 0.1711 - accuracy: 0.9483 - val_loss: 0.0697
- val_accuracy: 0.9780
Epoch 6/19
1875/1875 [==============================] - 89s 48ms/step - loss: 0.1580 - accuracy: 0.9528 - val_loss: 0.0750 -
val_accuracy: 0.9756
Epoch 7/19
1875/1875 [==============================] - 92s 49ms/step - loss: 0.1530 - accuracy: 0.9530 - val_loss: 0.1183 -
val_accuracy: 0.9640
Epoch 8/19
1875/1875 [==============================] - 94s 50ms/step - loss: 0.1431 - accuracy: 0.9560 - val_loss: 0.0402 -
val_accuracy: 0.9879
Epoch 9/19
1875/1875 [==============================] - 119s 63ms/step - loss: 0.1359 - accuracy: 0.9581 - val_loss: 0.0469
- val_accuracy: 0.9862
Epoch 10/19
1875/1875 [==============================] - 98s 52ms/step - loss: 0.1299 - accuracy: 0.9615 - val_loss: 0.0458 -
val_accuracy: 0.9859
Epoch 11/19
1875/1875 [==============================] - 102s 54ms/step - loss: 0.1260 - accuracy: 0.9626 - val_loss: 0.0632
- val_accuracy: 0.9811
Epoch 12/19
1875/1875 [==============================] - 114s 61ms/step - loss: 0.1205 - accuracy: 0.9634 - val_loss: 0.0430
- val_accuracy: 0.9870
Epoch 13/19
1875/1875 [==============================] - 116s 62ms/step - loss: 0.1151 - accuracy: 0.9656 - val_loss: 0.0443
- val_accuracy: 0.9869
Epoch 14/19
1875/1875 [==============================] - 119s 64ms/step - loss: 0.1151 - accuracy: 0.9647 - val_loss: 0.0476
- val_accuracy: 0.9842
Epoch 15/19
1875/1875 [==============================] - 93s 49ms/step - loss: 0.1102 - accuracy: 0.9664 - val_loss: 0.0445 -
val_accuracy: 0.9871
Epoch 16/19
1875/1875 [==============================] - 87s 46ms/step - loss: 0.1063 - accuracy: 0.9674 - val_loss: 0.0364 -
val_accuracy: 0.9885
Epoch 17/19
1875/1875 [==============================] - 86s 46ms/step - loss: 0.1052 - accuracy: 0.9668 - val_loss: 0.0440 -
val_accuracy: 0.9868
Epoch 18/19
1875/1875 [==============================] - 93s 49ms/step - loss: 0.1063 - accuracy: 0.9673 - val_loss: 0.0423 -
val_accuracy: 0.9864
Epoch 19/19
1875/1875 [==============================] - 96s 51ms/step - loss: 0.1038 - accuracy: 0.9686 - val_loss: 0.0405 -
val_accuracy: 0.9885
```

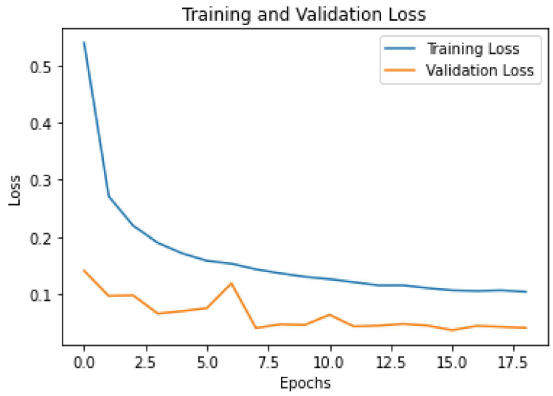
In [116]:

```
## checking for overfitting
'''
# Plot training and validation accuracy

'''

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history9_2.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history9_2.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With Batch Normalization)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [136]:

```
## checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history9_2.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history9_2.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [118]:

```
#make predictions
y_pred9_2 = np.argmax(cnn_model9_2.predict(x_test), axis=-1)
```

```
313/313 [==============================] - 4s 13ms/step
```

In [119]:

```
#Visualize the result
from sklearn.metrics import classification_report
cnn_classification_report9_3 = classification_report(y_test, y_pred9_2,output_dict=True)

print(classification_report(y_test, y_pred9_2))
```

```
              precision    recall  f1-score   support

           0       1.00      0.99      1.00       980
           1       1.00      1.00      1.00      1135
           2       0.97      0.97      0.97      1032
           3       0.99      1.00      1.00      1010
           4       0.99      1.00      0.99       982
           5       0.98      0.97      0.98       892
           6       0.98      0.98      0.98       958
           7       0.98      0.99      0.99      1028
           8       0.99      0.99      0.99       974
           9       1.00      0.98      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

In [120]:
```
df_report9_3 = pd.DataFrame(cnn_classification_report9_3).transpose()
df_report9_3
```

Out[120]:

	precision	recall	f1-score	support
0	0.998973	0.992857	0.995906	980.0000
1	0.998233	0.995595	0.996912	1135.0000
2	0.969171	0.974806	0.971981	1032.0000
3	0.993097	0.997030	0.995059	1010.0000
4	0.989899	0.997963	0.993915	982.0000
5	0.979661	0.971973	0.975802	892.0000
6	0.983281	0.982255	0.982768	958.0000
7	0.979866	0.994163	0.986963	1028.0000
8	0.994861	0.993840	0.994350	974.0000
9	0.996982	0.982161	0.989516	1009.0000
accuracy	0.988500	0.988500	0.988500	0.9885
macro avg	0.988402	0.988264	0.988317	10000.0000
weighted avg	0.988537	0.988500	0.988502	10000.0000

Observations:

- The model's accuracy decreased by 0.06% from 0.989100 to 0.988500 when the learning rate was changed to 0.0025 from 0.001 while the validation loss increased by 8.29% from 0.0374 to 0.0405.

Optimizer = SDG, Learning Rate = 0.0020, epoch = 19, and batch size = 32 (from my constant)

In [121]:
```
cnn_model9_3 = Sequential()

cnn_model9_3.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),padding = 'same'))
cnn_model9_3.add(BatchNormalization())
cnn_model9_3.add(Activation('relu'))
cnn_model9_3.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9_3.add(Conv2D(filters=64,kernel_size=(3,3)))
cnn_model9_3.add(BatchNormalization())
cnn_model9_3.add(Activation('relu'))
cnn_model9_3.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9_3.add(Conv2D(filters=128,kernel_size=(3,3)))
cnn_model9_3.add(BatchNormalization())
cnn_model9_3.add(Activation('relu'))
cnn_model9_3.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9_3.add(Flatten())
cnn_model9_3.add(BatchNormalization())
cnn_model9_3.add(Dense(128,activation = 'relu'))
cnn_model9_3.add(Dropout(0.5))
cnn_model9_3.add(Dense(10,activation = 'softmax'))

#compiling the model
sgd = SGD(learning_rate=0.0020, momentum=0.9)
cnn_model9_3.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
```

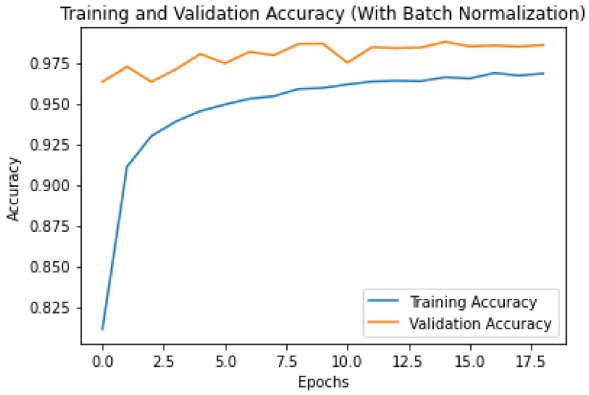
In [122]:

```
#Training and evaluating the model
history9_3 = cnn_model9_3.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_si
         epochs = 19, validation_data=(x_test, keras.utils.to_categorical(y_test)))
```

```
Epoch 1/19
1875/1875 [==============================] - 101s 53ms/step - loss: 0.5858 - accuracy: 0.8112 - val_loss: 0.1163
- val_accuracy: 0.9632
Epoch 2/19
1875/1875 [==============================] - 94s 50ms/step - loss: 0.2867 - accuracy: 0.9110 - val_loss: 0.0846 -
val_accuracy: 0.9726
Epoch 3/19
1875/1875 [==============================] - 94s 50ms/step - loss: 0.2276 - accuracy: 0.9298 - val_loss: 0.1135 -
val_accuracy: 0.9632
Epoch 4/19
1875/1875 [==============================] - 93s 50ms/step - loss: 0.1985 - accuracy: 0.9389 - val_loss: 0.0906 -
val_accuracy: 0.9710
Epoch 5/19
1875/1875 [==============================] - 96s 51ms/step - loss: 0.1789 - accuracy: 0.9453 - val_loss: 0.0629 -
val_accuracy: 0.9804
Epoch 6/19
1875/1875 [==============================] - 87s 46ms/step - loss: 0.1659 - accuracy: 0.9493 - val_loss: 0.0784 -
val_accuracy: 0.9746
Epoch 7/19
1875/1875 [==============================] - 85s 45ms/step - loss: 0.1546 - accuracy: 0.9528 - val_loss: 0.0559 -
val_accuracy: 0.9817
Epoch 8/19
1875/1875 [==============================] - 84s 45ms/step - loss: 0.1487 - accuracy: 0.9544 - val_loss: 0.0621 -
val_accuracy: 0.9796
Epoch 9/19
1875/1875 [==============================] - 95s 51ms/step - loss: 0.1358 - accuracy: 0.9588 - val_loss: 0.0453 -
val_accuracy: 0.9865
Epoch 10/19
1875/1875 [==============================] - 97s 51ms/step - loss: 0.1309 - accuracy: 0.9594 - val_loss: 0.0400 -
val_accuracy: 0.9867
Epoch 11/19
1875/1875 [==============================] - 95s 51ms/step - loss: 0.1292 - accuracy: 0.9616 - val_loss: 0.0803 -
val_accuracy: 0.9750
Epoch 12/19
1875/1875 [==============================] - 97s 52ms/step - loss: 0.1222 - accuracy: 0.9633 - val_loss: 0.0462 -
val_accuracy: 0.9845
Epoch 13/19
1875/1875 [==============================] - 107s 57ms/step - loss: 0.1182 - accuracy: 0.9639 - val_loss: 0.0489
- val_accuracy: 0.9840
Epoch 14/19
1875/1875 [==============================] - 100s 53ms/step - loss: 0.1179 - accuracy: 0.9636 - val_loss: 0.0468
- val_accuracy: 0.9843
Epoch 15/19
1875/1875 [==============================] - 98s 52ms/step - loss: 0.1126 - accuracy: 0.9661 - val_loss: 0.0392 -
val_accuracy: 0.9878
Epoch 16/19
1875/1875 [==============================] - 104s 56ms/step - loss: 0.1135 - accuracy: 0.9653 - val_loss: 0.0454
- val_accuracy: 0.9850
Epoch 17/19
1875/1875 [==============================] - 98s 52ms/step - loss: 0.1068 - accuracy: 0.9686 - val_loss: 0.0452 -
val_accuracy: 0.9855
Epoch 18/19
1875/1875 [==============================] - 97s 52ms/step - loss: 0.1055 - accuracy: 0.9671 - val_loss: 0.0459 -
val_accuracy: 0.9849
Epoch 19/19
1875/1875 [==============================] - 100s 53ms/step - loss: 0.1036 - accuracy: 0.9683 - val_loss: 0.0452
- val_accuracy: 0.9858
```

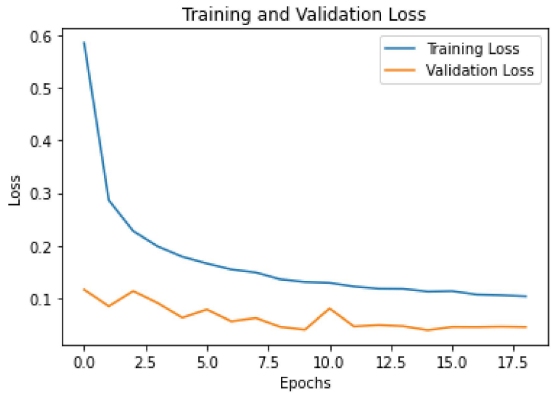
In [123]:

```
## checking for overfitting
'''
# Plot training and validation accuracy

'''

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history9_3.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history9_3.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With Batch Normalization)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [135]:

```
## checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history9_3.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history9_3.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [125]:

```
#make predictions
y_pred9_3 = np.argmax(cnn_model9_3.predict(x_test), axis=-1)
```

```
313/313 [==============================] - 4s 12ms/step
```

In [126]:

```
#Visualize the result
from sklearn.metrics import classification_report
cnn_classification_report9_4 = classification_report(y_test, y_pred9_3,output_dict=True)

print(classification_report(y_test, y_pred9_3))
```

```
              precision    recall  f1-score   support

           0       1.00      0.99      1.00       980
           1       1.00      1.00      1.00      1135
           2       0.97      0.98      0.97      1032
           3       0.98      1.00      0.99      1010
           4       0.99      1.00      0.99       982
           5       0.97      0.98      0.97       892
           6       0.99      0.97      0.98       958
           7       0.97      0.99      0.98      1028
           8       1.00      0.98      0.99       974
           9       1.00      0.97      0.98      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

In [127]:
```
df_report9_4 = pd.DataFrame(cnn_classification_report9_4).transpose()
df_report9_4
```

Out[127]:

	precision	recall	f1-score	support
0	0.998972	0.991837	0.995392	980.0000
1	0.995595	0.995595	0.995595	1135.0000
2	0.967402	0.977713	0.972530	1032.0000
3	0.984360	0.997030	0.990654	1010.0000
4	0.985915	0.997963	0.991903	982.0000
5	0.970000	0.978700	0.974330	892.0000
6	0.988335	0.972860	0.980537	958.0000
7	0.973282	0.992218	0.982659	1028.0000
8	0.997908	0.979466	0.988601	974.0000
9	0.995939	0.972250	0.983952	1009.0000
accuracy	0.985800	0.985800	0.985800	0.9858
macro avg	0.985771	0.985563	0.985615	10000.0000
weighted avg	0.985919	0.985800	0.985808	10000.0000

Observations:

- The model's accuracy decreased by 0.34% from 0.989100 to 0.985800 when the learning rate was changed to 0.0020 from 0.001 while the validation loss increased by 20.86% from 0.0374 to 0.0452.

Optimizer = SDG, Learning Rate = 0.0015, epoch = 19, and batch size = 32 (from my constant)

In [128]:
```
cnn_model9_4 = Sequential()

cnn_model9_4.add(Conv2D(filters=32,kernel_size=(3,3),input_shape=(28,28,1),padding = 'same'))
cnn_model9_4.add(BatchNormalization())
cnn_model9_4.add(Activation('relu'))
cnn_model9_4.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9_4.add(Conv2D(filters=64,kernel_size=(3,3)))
cnn_model9_4.add(BatchNormalization())
cnn_model9_4.add(Activation('relu'))
cnn_model9_4.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9_4.add(Conv2D(filters=128,kernel_size=(3,3)))
cnn_model9_4.add(BatchNormalization())
cnn_model9_4.add(Activation('relu'))
cnn_model9_4.add(MaxPooling2D(pool_size=(2,2)))

cnn_model9_4.add(Flatten())
cnn_model9_4.add(BatchNormalization())
cnn_model9_4.add(Dense(128,activation = 'relu'))
cnn_model9_4.add(Dropout(0.5))
cnn_model9_4.add(Dense(10,activation = 'softmax'))

#compiling the model
sgd = SGD(learning_rate=0.0015, momentum=0.9)
cnn_model9_4.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
```

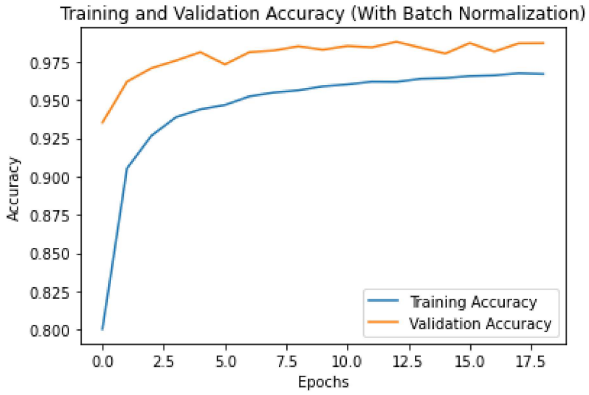
In [129]:

```
#Training and evaluating the model
history9_4 = cnn_model9_4.fit(train_datagenerator.flow(x_train, keras.utils.to_categorical(y_train), batch_si
        epochs = 19, validation_data=(x_test, keras.utils.to_categorical(y_test)))
```

```
Epoch 1/19
1875/1875 [==============================] - 93s 49ms/step - loss: 0.6176 - accuracy: 0.8003 - val_loss: 0.2148 -
val_accuracy: 0.9352
Epoch 2/19
1875/1875 [==============================] - 96s 51ms/step - loss: 0.2996 - accuracy: 0.9053 - val_loss: 0.1177 -
val_accuracy: 0.9619
Epoch 3/19
1875/1875 [==============================] - 96s 51ms/step - loss: 0.2348 - accuracy: 0.9266 - val_loss: 0.0902 -
val_accuracy: 0.9707
Epoch 4/19
1875/1875 [==============================] - 93s 50ms/step - loss: 0.2005 - accuracy: 0.9387 - val_loss: 0.0785 -
val_accuracy: 0.9757
Epoch 5/19
1875/1875 [==============================] - 97s 52ms/step - loss: 0.1827 - accuracy: 0.9438 - val_loss: 0.0571 -
val_accuracy: 0.9812
Epoch 6/19
1875/1875 [==============================] - 93s 50ms/step - loss: 0.1718 - accuracy: 0.9467 - val_loss: 0.0859 -
val_accuracy: 0.9732
Epoch 7/19
1875/1875 [==============================] - 114s 61ms/step - loss: 0.1579 - accuracy: 0.9523 - val_loss: 0.0575
- val_accuracy: 0.9812
Epoch 8/19
1875/1875 [==============================] - 124s 66ms/step - loss: 0.1476 - accuracy: 0.9548 - val_loss: 0.0567
- val_accuracy: 0.9823
Epoch 9/19
1875/1875 [==============================] - 115s 62ms/step - loss: 0.1430 - accuracy: 0.9562 - val_loss: 0.0457
- val_accuracy: 0.9850
Epoch 10/19
1875/1875 [==============================] - 118s 63ms/step - loss: 0.1346 - accuracy: 0.9588 - val_loss: 0.0547
- val_accuracy: 0.9828
Epoch 11/19
1875/1875 [==============================] - 105s 56ms/step - loss: 0.1307 - accuracy: 0.9601 - val_loss: 0.0476
- val_accuracy: 0.9852
Epoch 12/19
1875/1875 [==============================] - 97s 52ms/step - loss: 0.1248 - accuracy: 0.9619 - val_loss: 0.0504 -
val_accuracy: 0.9843
Epoch 13/19
1875/1875 [==============================] - 98s 52ms/step - loss: 0.1225 - accuracy: 0.9618 - val_loss: 0.0369 -
val_accuracy: 0.9879
Epoch 14/19
1875/1875 [==============================] - 110s 59ms/step - loss: 0.1200 - accuracy: 0.9638 - val_loss: 0.0501
- val_accuracy: 0.9841
Epoch 15/19
1875/1875 [==============================] - 95s 51ms/step - loss: 0.1169 - accuracy: 0.9643 - val_loss: 0.0625 -
val_accuracy: 0.9803
Epoch 16/19
1875/1875 [==============================] - 99s 53ms/step - loss: 0.1121 - accuracy: 0.9655 - val_loss: 0.0417 -
val_accuracy: 0.9871
Epoch 17/19
1875/1875 [==============================] - 97s 52ms/step - loss: 0.1118 - accuracy: 0.9660 - val_loss: 0.0563 -
val_accuracy: 0.9816
Epoch 18/19
1875/1875 [==============================] - 93s 50ms/step - loss: 0.1090 - accuracy: 0.9674 - val_loss: 0.0434 -
val_accuracy: 0.9869
Epoch 19/19
1875/1875 [==============================] - 98s 52ms/step - loss: 0.1091 - accuracy: 0.9669 - val_loss: 0.0378 -
val_accuracy: 0.9870
```

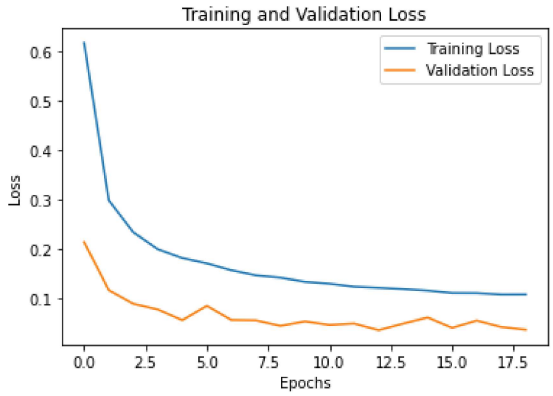
In [130]:

```
## checking for overfitting
'''
# Plot training and validation accuracy

'''

#Plots the training accuracy as a line with label "Training Accuracy".
plt.plot(history9_4.history['accuracy'], label='Training Accuracy')

#Plots the validation accuracy as a line with label "Validation Accuracy".
plt.plot(history9_4.history['val_accuracy'], label='Validation Accuracy')

#Sets the title of the plot to "Training and Validation Accuracy".
plt.title('Training and Validation Accuracy (With Batch Normalization)')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Accuracy".
plt.ylabel('Accuracy')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [134]:

```
## checking for overfitting

# Plot training loss against validation loss

#Plots the training loss as a line with label "Training Loss".
plt.plot(history9_4.history['loss'], label='Training Loss')

#Plots the validation loss as a line with label "Validation Loss".
plt.plot(history9_4.history['val_loss'], label='Validation Loss')

#Sets the title of the plot to "Training and Validation Loss".
plt.title('Training and Validation Loss')

#Sets the label of the x-axis to "Epochs".
plt.xlabel('Epochs')

#Sets the label of the y-axis to "Loss".
plt.ylabel('Loss')

#Shows the legend of the plot with the labels of the two lines.
plt.legend()

#Shows the plot on the screen.
plt.show()
```



In [139]:

```
#make predictions
y_pred9_4 = np.argmax(cnn_model9_4.predict(x_test), axis=-1)
```

```
313/313 [==============================] - 4s 13ms/step
```

In [140]:

```
#Visualize the result
from sklearn.metrics import classification_report
cnn_classification_report9_5 = classification_report(y_test, y_pred9_4,output_dict=True)

print(classification_report(y_test, y_pred9_4))
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       980
           1       1.00      1.00      1.00      1135
           2       0.97      0.97      0.97      1032
           3       0.99      1.00      0.99      1010
           4       0.98      1.00      0.99       982
           5       0.96      0.98      0.97       892
           6       0.99      0.98      0.98       958
           7       0.99      0.99      0.99      1028
           8       1.00      0.99      0.99       974
           9       0.99      0.98      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

In [141]:
```
df_report9_5 = pd.DataFrame(cnn_classification_report9_5).transpose()
df_report9_5
```

Out[141]:

	precision	recall	f1-score	support
0	0.993884	0.994898	0.994391	980.000
1	0.997354	0.996476	0.996915	1135.000
2	0.974659	0.968992	0.971817	1032.000
3	0.991124	0.995050	0.993083	1010.000
4	0.983920	0.996945	0.990389	982.000
5	0.959341	0.978700	0.968923	892.000
6	0.989451	0.979123	0.984260	958.000
7	0.987403	0.991245	0.989320	1028.000
8	0.996888	0.986653	0.991744	974.000
9	0.992972	0.980178	0.986534	1009.000
accuracy	0.987000	0.987000	0.987000	0.987
macro avg	0.986700	0.986826	0.986738	10000.000
weighted avg	0.987065	0.987000	0.987009	10000.000

Observations:

- The model's accuracy decreased by 0.21% from 0.989100 to 0.987000 when the learning rate was changed to 0.0015 from 0.001 while the validation loss increased by 1.0% from 0.0374 to 0.0378.
- Overall, the model had its highest accuracy of 0.991200 when the learning rate was changed to 0.005 from 0.001. This is an indication that a change in learning rate can affects a model's performance

Question(d)

Was there a case of overfitting observed in your model at any point ?

Observations:

- Using Adam optimizer there was a case of overfitting at the 6th epoch
- Also, overfitting was noticed at the 3rd, 4th, 6th, 7th, 9th, 10th, 11th, 13th and 14th epoch when RMSprop optimizer was used

In [ ]:
```

```