Component 2

In [1]:
```
#Importing Standard Libraries
import pandas as pd
import warnings
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from sklearn.metrics import v_measure_score, accuracy_score,adjusted_mutual_info_score,adjusted_rand_score,calinski_harabasz_sco
from sklearn.preprocessing import add_dummy_feature,Binarizer,LabelEncoder,LabelBinarizer,MinMaxScaler,OneHotEncoder,StandardSca
from sklearn.cluster import AgglomerativeClustering,DBSCAN,KMeans,spectral_clustering,dbscan,k_means
from sklearn.model_selection import train_test_split,cross_val_score,GridSearchCV,StratifiedGroupKFold,StratifiedKFold
from sklearn.ensemble import GradientBoostingClassifier,GradientBoostingRegressor,RandomForestClassifier,RandomForestRegressor,A
from sklearn.linear_model import Lasso,LinearRegression,LogisticRegression,Ridge,RidgeClassifier,SGDClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier,KNeighborsRegressor
from sklearn.naive_bayes import GaussianNB
warnings.filterwarnings("ignore")
```

In [2]:
```
#Loading the dataset
df=pd.read_csv('Video_Games.csv')
df.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
```

In [3]:
```
df.describe()
```

Out[3]:

	Year_of_Release	NA_Sales	EU_Sales	JP_Sales	Other_Sales	Global_Sales	Critic_Score	Critic_Count	User_Count
count	16450.000000	16719.000000	16719.000000	16719.000000	16719.000000	16719.000000	8137.000000	8137.000000	7590.000000
mean	2006.487356	0.263330	0.145025	0.077602	0.047332	0.533543	68.967679	26.360821	162.229908
std	5.878995	0.813514	0.503283	0.308818	0.186710	1.547935	13.938165	18.980495	561.282326
min	1980.000000	0.000000	0.000000	0.000000	0.000000	0.010000	13.000000	3.000000	4.000000
25%	2003.000000	0.000000	0.000000	0.000000	0.000000	0.060000	60.000000	12.000000	10.000000
50%	2007.000000	0.080000	0.020000	0.000000	0.010000	0.170000	71.000000	21.000000	24.000000
75%	2010.000000	0.240000	0.110000	0.040000	0.030000	0.470000	79.000000	36.000000	81.000000
max	2020.000000	41.360000	28.960000	10.220000	10.570000	82.530000	98.000000	113.000000	10665.000000

In [4]:
```
df.head(50)# Exploring the dataset-checking the head
```

Out[4]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sales	JP_Sales	Other_Sales	Global_Sales	Critic_Score	Critic_C
0	Wii Sports	Wii	2006.0	Sports	Nintendo	41.36	28.96	3.77	8.45	82.53	76.0	
1	Super Mario Bros.	NES	1985.0	Platform	Nintendo	29.08	3.58	6.81	0.77	40.24	NaN	
2	Mario Kart Wii	Wii	2008.0	Racing	Nintendo	15.68	12.76	3.79	3.29	35.52	82.0	
3	Wii Sports Resort	Wii	2009.0	Sports	Nintendo	15.61	10.93	3.28	2.95	32.77	80.0	
4	Pokemon Red/Pokemon Blue	GB	1996.0	Role-Playing	Nintendo	11.27	8.89	10.22	1.00	31.37	NaN	
5	Tetris	GB	1989.0	Puzzle	Nintendo	23.20	2.26	4.22	0.58	30.26	NaN	
6	New Super Mario Bros.	DS	2006.0	Platform	Nintendo	11.28	9.14	6.50	2.88	29.80	89.0	
7	Wii Play	Wii	2006.0	Misc	Nintendo	13.96	9.18	2.93	2.84	28.92	58.0	
8	New Super Mario Bros. Wii	Wii	2009.0	Platform	Nintendo	14.44	6.94	4.70	2.24	28.32	87.0	
9	Duck Hunt	NES	1984.0	Shooter	Nintendo	26.93	0.63	0.28	0.47	28.31	NaN	

In [5]:
```
df.columns#checking the number of columns
```

Out[5]:
```
Index(['Name', 'Platform', 'Year_of_Release', 'Genre', 'Publisher', 'NA_Sales',
       'EU_Sales', 'JP_Sales', 'Other_Sales', 'Global_Sales', 'Critic_Score',
       'Critic_Count', 'User_Score', 'User_Count', 'Developer', 'Rating'],
      dtype='object')
```

In [6]:
```
df.nunique().to_frame('Number of unique value')#Printing the Unique values in a dataframe
```

Out[6]:

	Number of unique value
Name	11562
Platform	31
Year_of_Release	39
Genre	12
Publisher	581
NA_Sales	402
EU_Sales	307
JP_Sales	244
Other_Sales	155
Global_Sales	629
Critic_Score	82
Critic_Count	106
User_Score	96
User_Count	888
Developer	1696
Rating	8

In [7]:
```
df.isnull().sum().to_frame('Number of null value')# printing the Number of null values
```

Out[7]:

	Number of null value
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

Data Cleaning

In [8]:
```
df = df.dropna(subset=['Publisher'])#dropping the null values in the publisher column
df = df.dropna(subset=['Genre'])#dropping the null values in the Genre column since the null entries are small
```

In [9]:
```
df.shape# checking the shape of the dataframe
```

Out[9]:
```
(16663, 16)
```

In [10]:
```
df['Year_of_Release'] = pd.to_datetime(df['Year_of_Release'], format='%Y')#converting to a date format
df['Year_of_Release'] = df['Year_of_Release'].dt.year
```

In [11]:
```
#function to extract year from the name column
import re# importing the regular expressions module
def fill_year_from_name(df):
    # Define regular expression to match year in game name
    year_format = r'\b(19|20)\d{2}\b'

    # Loop over rows and fill in missing year from name if possible
    for index, row in df.iterrows():
        name = row['Name']
        year = row['Year_of_Release']
        if pd.isna(year):
            match = re.search(year_format, name)
            if match:
                year = int(match.group(0))
                df.at[index, 'Year_of_Release'] = year

    return df
```

In [12]:
```
df = fill_year_from_name(df)# filling the year of release columns with the year from the name
```

In [13]:
```
median_year = int(df['Year_of_Release'].median())# filling the remaining missing values with the median of the years
df['Year_of_Release'] = df['Year_of_Release'].fillna(median_year).astype(int)
```

In [14]:
```
median_year
```

Out[14]:
```
2007
```

In [15]:
```
df.head(30)
```

Out[15]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sales	JP_Sales	Other_Sales	Global_Sales	Critic_Score	Critic_Coun
0	Wii Sports	Wii	2006	Sports	Nintendo	41.36	28.96	3.77	8.45	82.53	76.0	51.(
1	Super Mario Bros.	NES	1985	Platform	Nintendo	29.08	3.58	6.81	0.77	40.24	NaN	Nal
2	Mario Kart Wii	Wii	2008	Racing	Nintendo	15.68	12.76	3.79	3.29	35.52	82.0	73.(
3	Wii Sports Resort	Wii	2009	Sports	Nintendo	15.61	10.93	3.28	2.95	32.77	80.0	73.(
4	Pokemon Red/Pokemon Blue	GB	1996	Role-Playing	Nintendo	11.27	8.89	10.22	1.00	31.37	NaN	Nal
5	Tetris	GB	1989	Puzzle	Nintendo	23.20	2.26	4.22	0.58	30.26	NaN	Nal
6	New Super Mario Bros.	DS	2006	Platform	Nintendo	11.28	9.14	6.50	2.88	29.80	89.0	65.(
7	Wii Play	Wii	2006	Misc	Nintendo	13.96	9.18	2.93	2.84	28.92	58.0	41.(
8	New Super Mario Bros. Wii	Wii	2009	Platform	Nintendo	14.44	6.94	4.70	2.24	28.32	87.0	80.(
9	Duck Hunt	NES	1984	Shooter	Nintendo	26.93	0.63	0.28	0.47	28.31	NaN	Nal
10	Nintendogs	DS	2005	Simulation	Nintendo	9.05	10.95	1.93	2.74	24.67	NaN	Nal
11	Mario Kart DS	DS	2005	Racing	Nintendo	9.71	7.47	4.13	1.90	23.21	91.0	64.(
12	Pokemon Gold/Pokemon Silver	GB	1999	Role-Playing	Nintendo	9.00	6.18	7.20	0.71	23.10	NaN	Nal
13	Wii Fit	Wii	2007	Sports	Nintendo	8.92	8.03	3.60	2.15	22.70	80.0	63.(
14	Kinect Adventures!	X360	2010	Misc	Microsoft Game Studios	15.00	4.89	0.24	1.69	21.81	61.0	45.(
15	Wii Fit Plus	Wii	2009	Sports	Nintendo	9.01	8.49	2.53	1.77	21.79	80.0	33.(
16	Grand Theft Auto V	PS3	2013	Action	Take-Two Interactive	7.02	9.09	0.98	3.96	21.04	97.0	50.(
17	Grand Theft Auto: San Andreas	PS2	2004	Action	Take-Two Interactive	9.43	0.40	0.41	10.57	20.81	95.0	80.(
18	Super Mario World	SNES	1990	Platform	Nintendo	12.78	3.75	3.54	0.55	20.61	NaN	Nal
19	Brain Age: Train Your Brain in Minutes a Day	DS	2005	Misc	Nintendo	4.74	9.20	4.16	2.04	20.15	77.0	58.(
20	Pokemon Diamond/Pokemon Pearl	DS	2006	Role-Playing	Nintendo	6.38	4.46	6.04	1.36	18.25	NaN	Nal
21	Super Mario Land	GB	1989	Platform	Nintendo	10.83	2.71	4.18	0.42	18.14	NaN	Nal
22	Super Mario Bros. 3	NES	1988	Platform	Nintendo	9.54	3.44	3.84	0.46	17.28	NaN	Nal
23	Grand Theft Auto V	X360	2013	Action	Take-Two Interactive	9.66	5.14	0.06	1.41	16.27	97.0	58.(
24	Grand Theft Auto: Vice City	PS2	2002	Action	Take-Two Interactive	8.41	5.49	0.47	1.78	16.15	95.0	62.(
25	Pokemon Ruby/Pokemon Sapphire	GBA	2002	Role-Playing	Nintendo	6.06	3.90	5.38	0.50	15.85	NaN	Nal
26	Brain Age 2: More Training in Minutes a Day	DS	2005	Puzzle	Nintendo	3.43	5.35	5.32	1.18	15.29	77.0	37.(
27	Pokemon Black/Pokemon White	DS	2010	Role-Playing	Nintendo	5.51	3.17	5.65	0.80	15.14	NaN	Nal
28	Gran Turismo 3: A-Spec	PS2	2001	Racing	Sony Computer Entertainment	6.85	5.09	1.87	1.16	14.98	95.0	54.(
29	Call of Duty: Modern Warfare 3	X360	2011	Shooter	Activision	9.04	4.24	0.13	1.32	14.73	88.0	81.(

In [16]:
```
#Creating a dataframe for feature extraction for filling missing values
df_critic_score=df.drop(['Name', 'Platform','Genre', 'Publisher','Critic_Count' ,'User_Score', 'User_Count', 'Developer', 'Ratin
df_critic_count=df.drop(['Name', 'Platform','Genre', 'Publisher','Critic_Score' ,'User_Score', 'User_Count', 'Developer', 'Ratin
df_user_score=df.drop(['Name', 'Platform','Genre', 'Publisher','Critic_Count' ,'Critic_Score', 'User_Count', 'Developer', 'Ratin
df_user_count=df.drop(['Name', 'Platform','Genre', 'Publisher','Critic_Count' ,'Critic_Score', 'User_Score', 'Developer', 'Ratin
df_rating=df.drop(['Name', 'Platform','Genre', 'Publisher','Critic_Count' ,'Critic_Score', 'User_Score','User_Count', 'Developer
```

In [17]:
```
df.isnull().sum().to_frame('Number of null value')
```

Out[17]:

	Number of null value
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	8530
Critic_Count	8530
User_Score	6656
User_Count	9079
Developer	6576
Rating	6720

Data Cleaning-using regression analysis

In [18]:
```
#Feature selection to predict critic score
df_pred=df.dropna() #Creating a subset from the dataframe without null values
```

In [19]:
```
df_pred.info()# Checking the info of the dataset
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6943 entries, 0 to 16706
Data columns (total 16 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   Name             6943 non-null   object 
 1   Platform         6943 non-null   object 
 2   Year_of_Release  6943 non-null   int32  
 3   Genre            6943 non-null   object 
 4   Publisher        6943 non-null   object 
 5   NA_Sales         6943 non-null   float64
 6   EU_Sales         6943 non-null   float64
 7   JP_Sales         6943 non-null   float64
 8   Other_Sales      6943 non-null   float64
 9   Global_Sales     6943 non-null   float64
 10  Critic_Score     6943 non-null   float64
 11  Critic_Count     6943 non-null   float64
 12  User_Score       6943 non-null   object 
 13  User_Count       6943 non-null   float64
 14  Developer        6943 non-null   object 
 15  Rating           6943 non-null   object 
dtypes: float64(8), int32(1), object(7)
memory usage: 895.0+ KB
```

In [20]:
```
df_pred.shape
```

Out[20]: (6943, 16)

In [21]:
```
#feature extraction
X_pred= df_pred.drop(['Name', 'Platform','Genre', 'Publisher','Critic_Score','Critic_Count', 'User_Score', 'User_Count', 'Develo
```

In [22]:
```
#Predicting for the critic score
X=X_pred
y=df_pred['Critic_Score']
```

In [23]:
```
X.shape
```

Out[23]: (6943, 6)

In [24]:
```
y.shape
```

Out[24]: (6943,)

In [25]:
```
# Define the input and output features
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
```

In [26]:
```
X_test.shape
```

Out[26]: (1389, 6)

In [27]:
```
#Normalising dataset
scale=StandardScaler()#initialising Normaliser
scale.fit(X_train)#fitting Normaliser to train data
X_train_scaled=scale.transform(X_train)
X_test_scaled=scale.transform(X_test)
```

In [28]:
```
# Train the model
model = LinearRegression()
model.fit(X_train_scaled, y_train)

# Predict the missing values
y_pred = model.predict(X_test_scaled)
print("Linear Regression MSE :", mean_squared_error(y_test, y_pred))
print("Linear Regression R2 score :", r2_score(y_test, y_pred))
# Calculate the root mean squared error (RMSE) on the training set
y_train_pred = model.predict(X_train_scaled)
rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training RMSE:', rmse)
```

```
Linear Regression MSE : 178.80385451487052
Linear Regression R2 score : 0.08040379070418557
Training R2: 0.050716584534168496
Training RMSE: 13.51057823967585
```

In [29]:
```
kn=KNeighborsRegressor(n_neighbors=5,weights='distance')
kn.fit(X_train_scaled,y_train)
kn_pred=kn.predict(X_test_scaled)
print("K Neighbours Regression MSE :", mean_squared_error(y_test, kn_pred))
print("KNeighbours R2 score :", r2_score(y_test, y_pred))
y_train_pred = kn.predict(X_train_scaled)
rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training RMSE:', rmse)
```

```
K Neighbours Regression MSE : 187.16020958987994
KNeighbours R2 score : 0.08040379070418557
Training R2: 0.8491428963286171
Training RMSE: 5.38590728097198
```

In [30]:
```
rf_model =RandomForestRegressor(n_estimators=100,random_state=42)
rf_model.fit(X_train_scaled, y_train)
# Predict the missing values
y_pred = rf_model.predict(X_test_scaled)
print("Random forest Regression MSE :", mean_squared_error(y_test, y_pred))
print("Random forest R2 score :", r2_score(y_test, y_pred))
# Calculate the root mean squared error (RMSE) on the training set
y_train_pred = rf_model.predict(X_train_scaled)
rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training RMSE:', rmse)
```

```
Random forest Regression MSE : 166.85622807923534
Random forest R2 score : 0.14185096705338895
Training R2: 0.7704620538021185
Training RMSE: 6.643596318252831
```

In [31]:
```
#Since Random forest regressor provided the best prediction, the model will be used to predict the critic scores
mask = df_critic_score['Critic_Score'].isnull()
X_missing = df_critic_score[mask].drop('Critic_Score', axis=1)
X_missing_scaled = scale.transform(X_missing)
y_missing_pred = rf_model.predict(X_missing_scaled)
df_critic_score.loc[mask, 'Critic_Score'] = y_missing_pred
```

In [32]:
```
df_critic_score['Critic_Score']# checking the new critic score series created
```

Out[32]:
```
0        76.000000
1        86.170000
2        82.000000
3        80.000000
4        84.910000
           ...
16714    69.352110
16715    70.466839
16716    69.352110
16717    49.540833
16718    69.352110
Name: Critic_Score, Length: 16663, dtype: float64
```

In [33]:
```
df['Critic_Score']=df_critic_score['Critic_Score']# replacing with the original dataframe
```

In [34]:
```
#Predicitng the missing values for the Critic count
X=X_pred
y=df_pred['Critic_Count']
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
#Normalising dataset
scale=StandardScaler()#initialising Normaliser
scale.fit(X_train)#fitting Normaliser to train data
X_train_scaled=scale.transform(X_train)
X_test_scaled=scale.transform(X_test)

rf_model =RandomForestRegressor(n_estimators=100,random_state=42)
rf_model.fit(X_train_scaled, y_train)
# Predict the missing values
y_pred = rf_model.predict(X_test_scaled)
print("Random forest Regression MSE :", mean_squared_error(y_test, y_pred))
print("Random forest R2 score :", r2_score(y_test, y_pred))
# Calculate the root mean squared error (RMSE) on the training set
y_train_pred = rf_model.predict(X_train_scaled)
rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training RMSE:', rmse)
```

```
Random forest Regression MSE : 245.19744448134
Random forest R2 score : 0.36209148871183583
Training R2: 0.8661422378144845
Training RMSE: 6.985424590241286
```

In [35]:
```
#Since Random forest regressor provided the best prediction, this will be used to predict the critic count
mask = df_critic_count['Critic_Count'].isnull()
X_missing = df_critic_count[mask].drop('Critic_Count', axis=1)
X_missing_scaled = scale.transform(X_missing)
y_missing_pred = rf_model.predict(X_missing_scaled)
df_critic_count.loc[mask, 'Critic_Count'] = y_missing_pred
```

In [36]:
```
df['Critic_Count']=df_critic_count['Critic_Count']
```

In [37]:
```
# Replacing 'tbd' values with NaN
df_user_score['User_Score'] = df_user_score['User_Score'].replace('tbd', np.nan)
```

In [38]:
```
#Predicting values in the User score column
X=X_pred
y=df_pred['User_Score']
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
#Normalising dataset
scale=StandardScaler()#initialising Normaliser
scale.fit(X_train)#fitting Normaliser to train data
X_train_scaled=scale.transform(X_train)
X_test_scaled=scale.transform(X_test)

rf_model =RandomForestRegressor(n_estimators=100,random_state=42)
rf_model.fit(X_train_scaled, y_train)
# Predict the missing values
y_pred = rf_model.predict(X_test_scaled)
print("Random forest Regression MSE :", mean_squared_error(y_test, y_pred))
print("Random forest R2 score :", r2_score(y_test, y_pred))
# Calculate the root mean squared error (RMSE) on the training set
y_train_pred = rf_model.predict(X_train_scaled)
rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training RMSE:', rmse)
```

```
Random forest Regression MSE : 2.1197023070280685
Random forest R2 score : 0.006334064760603164
Training R2: 0.7257351332214976
Training RMSE: 0.7519551667174884
```

In [39]:
```
mask = df_user_score['User_Score'].isnull()
X_missing = df_user_score[mask].drop('User_Score', axis=1)
X_missing_scaled = scale.transform(X_missing)
y_missing_pred = rf_model.predict(X_missing_scaled)
df_user_score.loc[mask, 'User_Score'] = y_missing_pred
```

In [40]:
```
df['User_Score']=df_user_score['User_Score']# replacing with the original dataframe
```

In [41]:
```
X=X_pred
y=df_pred['User_Count']
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
#Normalising dataset
scale=StandardScaler()#initialising Normaliser
scale.fit(X_train)#fitting Normaliser to train data
X_train_scaled=scale.transform(X_train)
X_test_scaled=scale.transform(X_test)

rf_model =RandomForestRegressor(n_estimators=100,random_state=42)
rf_model.fit(X_train_scaled, y_train)
# Predict the missing values
y_pred = rf_model.predict(X_test_scaled)
print("Random forest Regression MSE :", mean_squared_error(y_test, y_pred))
print("Random forest R2 score :", r2_score(y_test, y_pred))
# Calculate the root mean squared error (RMSE) on the training set
y_train_pred = rf_model.predict(X_train_scaled)
rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training RMSE:', rmse)
```

```
Random forest Regression MSE : 207578.00793170574
Random forest R2 score : 0.35689646711960865
Training R2: 0.8752623096835777
Training RMSE: 207.9075203885757
```

In [42]:
```
mask = df_user_count['User_Count'].isnull()
X_missing = df_user_count[mask].drop('User_Count', axis=1)
X_missing_scaled = scale.transform(X_missing)
y_missing_pred = rf_model.predict(X_missing_scaled)
df_user_count.loc[mask, 'User_Count'] = y_missing_pred
```

In [43]:
```
df['User_Count']=df_user_count['User_Count']
```

In [44]:
```
mapping_dict = df.groupby(['Genre', 'Publisher'])['Developer'].apply(lambda x: x.mode()[0] if not x.mode().empty else None).to_d

# Define a function to updated developer column based on mapping_dict
def updated_developer(row):
    if pd.isna(row['Developer']):
        return row['Publisher']
    else:
        return row['Developer']

# Apply the function to update the Developer column
df['Developer'] = df.apply(updated_developer, axis=1)
```

Predicting the missing entries in the rating column

In [45]:
```
#Using KNeighbors classifier
#predicting values for the rating column
X=X_pred
y=df_pred['Rating']
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)

#Normalising dataset
scale=StandardScaler()#initialising Normaliser
scale.fit(X_train)#fitting Normaliser to train data
X_train_scaled=scale.transform(X_train)
X_test_scaled=scale.transform(X_test)

#Instatiating and fitting the model
kn_clf=KNeighborsClassifier(n_neighbors=5,weights='distance',algorithm='auto')
kn_clf.fit(X_train_scaled, y_train)
# Predict the missing values
y_pred=kn_clf.predict(X_test_scaled)
print('Classification Report \n')
print(classification_report(y_test,y_pred))
print('Confusion Matrix \n')
print(confusion_matrix(y_test,y_pred))
```

```
Classification Report

              precision    recall  f1-score   support

          AO       0.00      0.00      0.00         1
           E       0.43      0.44      0.44       418
        E10+       0.27      0.23      0.25       184
           M       0.37      0.36      0.36       314
           T       0.42      0.44      0.43       472

    accuracy                           0.40      1389
   macro avg       0.30      0.30      0.30      1389
weighted avg       0.39      0.40      0.39      1389

Confusion Matrix

[[  0   0   0   0   1]
 [  0 186  39  65 128]
 [  0  44  43  39  58]
 [  0  74  33 113  94]
 [  0 131  44  90 207]]
```

In [ ]:
```
#Using Logistic Regression
logreg_clf=LogisticRegression()
grid_param= {
    'penalty': ['l1','l2'],
    'solver':['newton-cg','lbfgs','liblinear']
}
#fitting the GCSV model to the training dataset
gcsv=GridSearchCV(logreg_clf,grid_param,scoring='accuracy',cv=5)
gcsv.fit(X_train_scaled, y_train)
```

In [47]:
```
print("Tuned Hyperparameters :",gcsv.best_params_)
print("Accuracy :",gcsv.best_score_)
```

```
Tuned Hyperparameters : {'penalty': 'l2', 'solver': 'newton-cg'}
Accuracy : 0.38747090925308747
```

In [48]:
```
#Using the best parameters {'penalty': 'l2', 'solver': 'newton-cg'}
logreg_clf=LogisticRegression(C=0.1,penalty='l2',solver='newton-cg')
logreg_clf.fit(X_train_scaled, y_train)
logreg_pred=logreg_clf.predict(X_test_scaled)
print('Classification Report \n')
print(classification_report(y_test,logreg_pred))
print('Confusion Matrix \n')
print(confusion_matrix(y_test,logreg_pred))
```

```
Classification Report

              precision    recall  f1-score   support

          AO       0.00      0.00      0.00         1
           E       0.42      0.23      0.30       418
        E10+       0.00      0.00      0.00       184
           M       0.41      0.26      0.32       314
           T       0.37      0.74      0.49       472

    accuracy                           0.38      1389
   macro avg       0.24      0.25      0.22      1389
weighted avg       0.34      0.38      0.33      1389

Confusion Matrix

[[  0   1   0   0   0]
 [  0  98   0  40 280]
 [  0  12   0  32 140]
 [  0  44   0  82 188]
 [  0  76   0  46 350]]
```

In [49]:
```
#Using Random Forest Classifier
#Instatiating and fitting the model
rf_clf=RandomForestClassifier(n_estimators=100)
rf_clf.fit(X_train_scaled, y_train)
# Predict the missing values
rf_pred=rf_clf.predict(X_test_scaled)
print('Classification Report \n')
print(classification_report(y_test,rf_pred))
print('Confusion Matrix \n')
print(confusion_matrix(y_test,rf_pred))
```

```
Classification Report

              precision    recall  f1-score   support

          AO       0.00      0.00      0.00         1
           E       0.45      0.47      0.46       418
        E10+       0.23      0.17      0.20       184
           M       0.38      0.34      0.36       314
           T       0.41      0.46      0.44       472

    accuracy                           0.40      1389
   macro avg       0.29      0.29      0.29      1389
weighted avg       0.39      0.40      0.39      1389

Confusion Matrix

[[  0   0   0   0   1]
 [  0 197  36  55 130]
 [  0  46  32  33  73]
 [  0  69  32 108 105]
 [  0 125  42  87 218]]
```

In [50]:
```
#since randomforestclassifier produces a better accuracy, Random forest classifer will be used for the prediction of the rating
# Fill in the missing values in the 'Rating' column with the predicted values
mask = df_rating['Rating'].isnull()
X_missing = df_rating[mask].drop('Rating', axis=1)
X_missing_scaled = scale.transform(X_missing)
y_missing_pred = rf_clf.predict(X_missing_scaled)
df_rating.loc[mask, 'Rating'] = y_missing_pred
```

In [51]:
```
df['Rating']=df_rating['Rating'] # replacing the rating column with the rating predictions without affecting the original dataset
```

In [52]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16663 entries, 0 to 16662
Data columns (total 16 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Name             16663 non-null  object
 1   Platform         16663 non-null  object
 2   Year_of_Release  16663 non-null  int32
 3   Genre            16663 non-null  object
 4   Publisher        16663 non-null  object
 5   NA_Sales         16663 non-null  float64
 6   EU_Sales         16663 non-null  float64
 7   JP_Sales         16663 non-null  float64
 8   Other_Sales      16663 non-null  float64
 9   Global_Sales     16663 non-null  float64
 10  Critic_Score     16663 non-null  float64
 11  Critic_Count     16663 non-null  float64
 12  User_Score       16663 non-null  object
 13  User_Count       16663 non-null  float64
 14  Developer        16663 non-null  object
 15  Rating           16663 non-null  object
dtypes: float64(8), int32(1), object(7)
memory usage: 2.6+ MB
```

In [53]:
```
df.head(30)# checking the head of the new dataframe
```

Out[53]:

	Name	Platform	Year_of_Release	Genre	Publisher	NA_Sales	EU_Sales	JP_Sales	Other_Sales	Global_Sales	Critic_Score	Critic_Coun
0	Wii Sports	Wii	2006	Sports	Nintendo	41.36	28.96	3.77	8.45	82.53	76.00	51.0
1	Super Mario Bros.	NES	1985	Platform	Nintendo	29.08	3.58	6.81	0.77	40.24	86.17	20.3
2	Mario Kart Wii	Wii	2008	Racing	Nintendo	15.68	12.76	3.79	3.29	35.52	82.00	73.0
3	Wii Sports Resort	Wii	2009	Sports	Nintendo	15.61	10.93	3.28	2.95	32.77	80.00	73.0
4	Pokemon Red/Pokemon Blue	GB	1996	Role-Playing	Nintendo	11.27	8.89	10.22	1.00	31.37	84.91	21.1
5	Tetris	GB	1989	Puzzle	Nintendo	23.20	2.26	4.22	0.58	30.26	85.09	20.2
6	New Super Mario Bros.	DS	2006	Platform	Nintendo	11.28	9.14	6.50	2.88	29.80	89.00	65.0
7	Wii Play	Wii	2006	Misc	Nintendo	13.96	9.18	2.93	2.84	28.92	58.00	41.0
8	New Super Mario Bros. Wii	Wii	2009	Platform	Nintendo	14.44	6.94	4.70	2.24	28.32	87.00	80.0
9	Duck Hunt	NES	1984	Shooter	Nintendo	26.93	0.63	0.28	0.47	28.31	87.86	19.0
10	Nintendogs	DS	2005	Simulation	Nintendo	9.05	10.95	1.93	2.74	24.67	71.49	52.9
11	Mario Kart DS	DS	2005	Racing	Nintendo	9.71	7.47	4.13	1.90	23.21	91.00	64.0
12	Pokemon Gold/Pokemon Silver	GB	1999	Role-Playing	Nintendo	9.00	6.18	7.20	0.71	23.10	88.29	25.5
13	Wii Fit	Wii	2007	Sports	Nintendo	8.92	8.03	3.60	2.15	22.70	80.00	63.0
14	Kinect Adventures!	X360	2010	Misc	Microsoft Game Studios	15.00	4.89	0.24	1.69	21.81	61.00	45.0
15	Wii Fit Plus	Wii	2009	Sports	Nintendo	9.01	8.49	2.53	1.77	21.79	80.00	33.0
16	Grand Theft Auto V	PS3	2013	Action	Take-Two Interactive	7.02	9.09	0.98	3.96	21.04	97.00	50.0
17	Grand Theft Auto: San Andreas	PS2	2004	Action	Take-Two Interactive	9.43	0.40	0.41	10.57	20.81	95.00	80.0
18	Super Mario World	SNES	1990	Platform	Nintendo	12.78	3.75	3.54	0.55	20.61	87.73	19.9
19	Brain Age: Train Your Brain in Minutes a Day	DS	2005	Misc	Nintendo	4.74	9.20	4.16	2.04	20.15	77.00	58.0
20	Pokemon Diamond/Pokemon Pearl	DS	2006	Role-Playing	Nintendo	6.38	4.46	6.04	1.36	18.25	88.88	60.7
21	Super Mario Land	GB	1989	Platform	Nintendo	10.83	2.71	4.18	0.42	18.14	89.56	20.0
22	Super Mario Bros. 3	NES	1988	Platform	Nintendo	9.54	3.44	3.84	0.46	17.28	88.70	20.0
23	Grand Theft Auto V	X360	2013	Action	Take-Two Interactive	9.66	5.14	0.06	1.41	16.27	97.00	58.0
24	Grand Theft Auto: Vice City	PS2	2002	Action	Take-Two Interactive	8.41	5.49	0.47	1.78	16.15	95.00	62.0
25	Pokemon Ruby/Pokemon Sapphire	GBA	2002	Role-Playing	Nintendo	6.06	3.90	5.38	0.50	15.85	88.27	46.5
26	Brain Age 2: More Training in Minutes a Day	DS	2005	Puzzle	Nintendo	3.43	5.35	5.32	1.18	15.29	77.00	37.0
27	Pokemon Black/Pokemon White	DS	2010	Role-Playing	Nintendo	5.51	3.17	5.65	0.80	15.14	87.21	75.6
28	Gran Turismo 3: A-Spec	PS2	2001	Racing	Sony Computer Entertainment	6.85	5.09	1.87	1.16	14.98	95.00	54.0
29	Call of Duty: Modern Warfare 3	X360	2011	Shooter	Activision	9.04	4.24	0.13	1.32	14.73	88.00	81.0

In [54]:
```
#creating a mapping dictionary for the developer based on Genre and Publisher using lambda functions
mapping_dict = df.groupby(['Publisher', 'Genre'])['Developer'].apply(lambda x: x.mode()[0] if not x.mode().empty else None).to_d
#Function to replace null values in developer column with the publisher column and also considering the Genre
def updated_developer(row):
    if pd.isna(row['Developer']):
        key = (row['Publisher'], row['Genre'])
        if key in mapping_dict:
            return mapping_dict[key]
        elif pd.notna(row['Publisher']):
            return row['Publisher']
        elif pd.notna(row['Genre']):
            return row['Genre']
        else:
            return None
    elif isinstance(row['Developer'], (str, list)):
        return row['Developer']
    else:
        return mapping_dict.get((row['Publisher'], row['Genre']), row['Publisher'])

df['Developer'] = df.apply(update_developer, axis=1)#applying the updated function into the developer column
```

In [55]:
```
df.isnull().sum().to_frame('Number of null value')# checking the null values in the cleaned dataset
```

Out[55]:

	Number of null value
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

In [56]:
```
df['User_Score']=df['User_Score'].astype(float)
```

In [57]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 16663 entries, 0 to 16718
Data columns (total 16 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Name             16663 non-null  object
 1   Platform         16663 non-null  object
 2   Year_of_Release  16663 non-null  int32
 3   Genre            16663 non-null  object
 4   Publisher        16663 non-null  object
 5   NA_Sales         16663 non-null  float64
 6   EU_Sales         16663 non-null  float64
 7   JP_Sales         16663 non-null  float64
 8   Other_Sales      16663 non-null  float64
 9   Global_Sales     16663 non-null  float64
 10  Critic_Score     16663 non-null  float64
 11  Critic_Count     16663 non-null  float64
 12  User_Score       16663 non-null  float64
 13  User_Count       16663 non-null  float64
 14  Developer        16663 non-null  object
 15  Rating           16663 non-null  object
dtypes: float64(9), int32(1), object(6)
memory usage: 2.6+ MB
```

In [58]:
```
new_df=df.copy()
```

Using Regression Analysis to determine the best features to predict the global sales

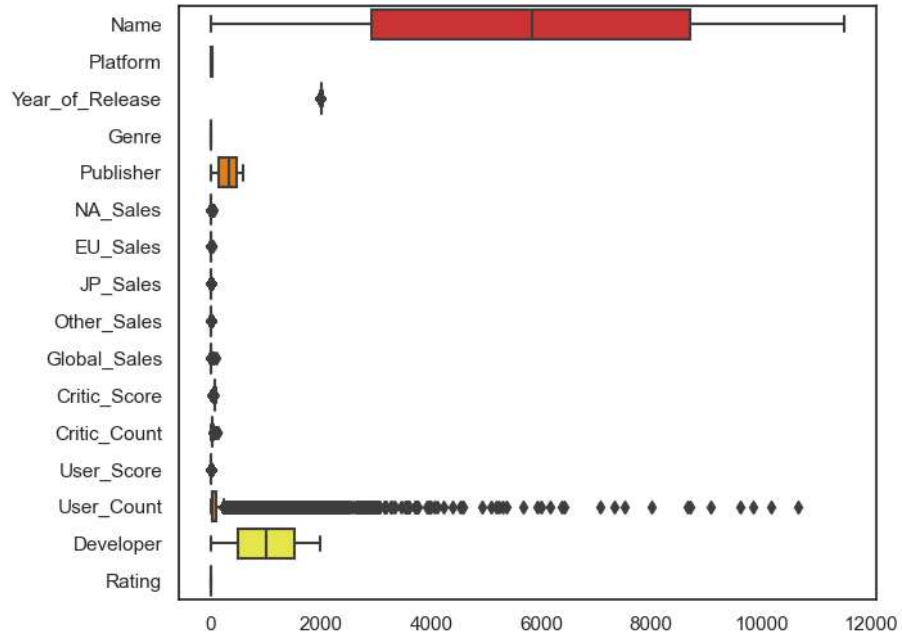
In [59]:
```
# Select the categorical columns to encode
cat_columns = ['Name', 'Platform', 'Genre', 'Publisher', 'Developer', 'Rating']

# Create a LabelEncoder instance for each categorical column and fit-transform
for column in cat_columns:
    le = LabelEncoder()
    df[column] = le.fit_transform(df[column].astype(str))
```

In [60]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 16663 entries, 0 to 16718
Data columns (total 16 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   Name             16663 non-null  int32  
 1   Platform         16663 non-null  int32  
 2   Year_of_Release  16663 non-null  int32  
 3   Genre            16663 non-null  int32  
 4   Publisher        16663 non-null  int32  
 5   NA_Sales         16663 non-null  float64
 6   EU_Sales         16663 non-null  float64
 7   JP_Sales         16663 non-null  float64
 8   Other_Sales      16663 non-null  float64
 9   Global_Sales     16663 non-null  float64
 10  Critic_Score     16663 non-null  float64
 11  Critic_Count     16663 non-null  float64
 12  User_Score       16663 non-null  float64
 13  User_Count       16663 non-null  float64
 14  Developer        16663 non-null  int32  
 15  Rating           16663 non-null  int32  
dtypes: float64(9), int32(7)
memory usage: 2.2 MB
```

In [173]:
```
#Box plot to show feature distribution and the need to normalise dataset so as to take away bias
fig, ax = plt.subplots(figsize=(7,6))
sns.boxplot(data=df,orient='h',palette='Set1')
```

Out[173]:
```
<AxesSubplot:>
```



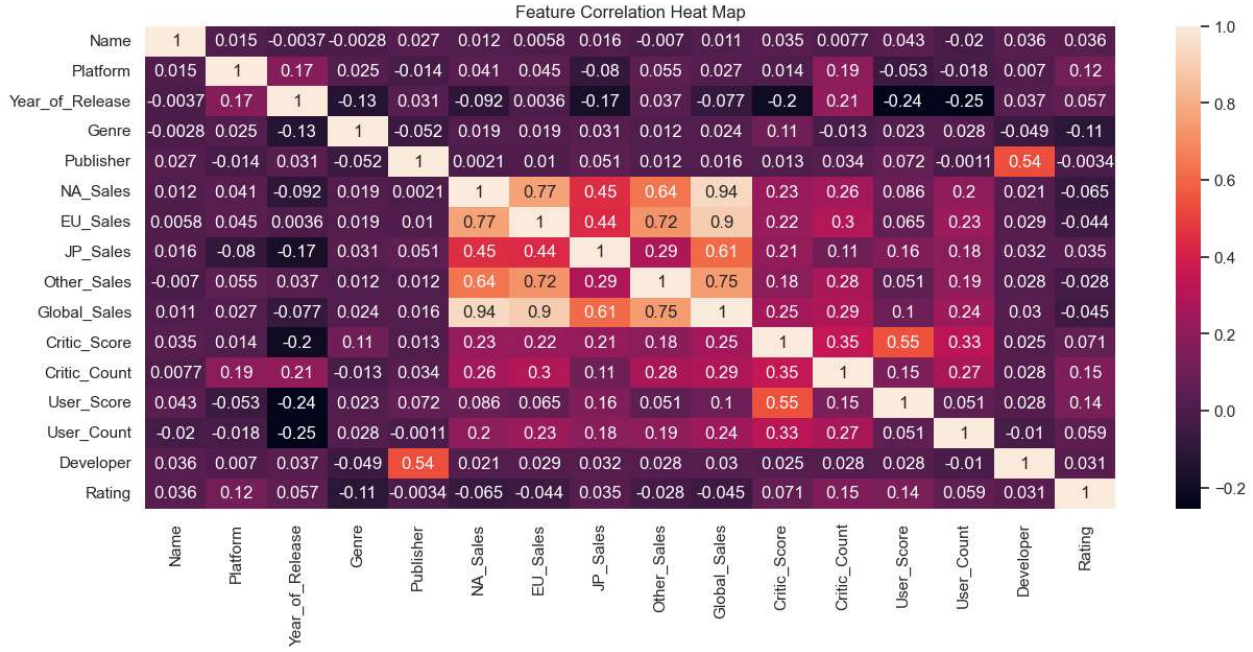
In [61]:
```
df.columns
```

Out[61]:
```
Index(['Name', 'Platform', 'Year_of_Release', 'Genre', 'Publisher', 'NA_Sales',
       'EU_Sales', 'JP_Sales', 'Other_Sales', 'Global_Sales', 'Critic_Score',
       'Critic_Count', 'User_Score', 'User_Count', 'Developer', 'Rating'],
      dtype='object')
```

In [62]:
```
df.head(30)
```

18	9367	23	1990	4	361	12.78	3.75	3.54	0.55	20.61	87.73	19.98	7.925
19	1019	4	2005	3	361	4.74	9.20	4.16	2.04	20.15	77.00	58.00	7.900
20	7350	4	2006	7	361	6.38	4.46	6.04	1.36	18.25	88.88	60.77	8.157
21	9360	5	1989	4	361	10.83	2.71	4.18	0.42	18.14	89.56	20.04	7.998
22	9355	11	1988	4	361	9.54	3.44	3.84	0.46	17.28	88.70	20.04	7.987
23	3722	28	2013	0	497	9.66	5.14	0.06	1.41	16.27	97.00	58.00	8.100
24	3727	16	2002	0	497	8.41	5.49	0.47	1.78	16.15	95.00	62.00	8.700
25	7366	6	2002	7	361	6.06	3.90	5.38	0.50	15.85	88.27	46.53	8.470
26	1017	4	2005	5	361	3.43	5.35	5.32	1.18	15.29	77.00	37.00	7.100
27	7344	4	2010	7	361	5.51	3.17	5.65	0.80	15.14	87.21	75.63	8.071
28	3703	16	2001	6	458	6.85	5.09	1.87	1.16	14.98	95.00	54.00	8.400
29	1243	28	2011	8	21	9.04	4.24	0.13	1.32	14.73	88.00	81.00	3.400

In [149]:
```
#plotting the feature correlation heatmap
fig, ax = plt.subplots(figsize=(15,6))
sns.set_style('white')
sns.heatmap(df.corr(),annot=True);
ax.set_title('Feature Correlation Heat Map')
fig.savefig("Correlation Heat Map.png", dpi=300)
```



In [64]:
```
#Feature selection-picking the variables that significantly contribute to the Global sales
#from the heatmap above it is evident that the NA sales,EU sales ,JP sales and other sales contributes significantly to the glob
X=df[['NA_Sales','EU_Sales','JP_Sales', 'Other_Sales','Critic_Count','User_Count','Platform','Genre', 'Developer']]
y=df['Global_Sales']
# Define the input and output features
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
#Normalising dataset
scale=StandardScaler()#initialising Normaliser
scale.fit(X_train)#fitting Normaliser to train data
X_train_scaled=scale.transform(X_train)
X_test_scaled=scale.transform(X_test)
```

In [65]:
```
# Train the model using Linear regression
lm = LinearRegression()
lm.fit(X_train_scaled, y_train)

# Predict the missing values
y_pred = lm.predict(X_test_scaled)
print("Linear Regression MAE :", mean_absolute_error(y_test, y_pred))
print("Linear Regression MSE :", mean_squared_error(y_test, y_pred))
print("Linear Regression R2 :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = lm.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
Linear Regression MAE : 0.0029135768947378318
Linear Regression MSE : 2.730676458277518e-05
Linear Regression R2 : 0.9999934347507479
Training R2: 0.9999861788248636
Training MAE: 0.0028916089058617055
Training MSE: 2.713516806394461e-05
```

In [66]:
```
# Train the model using KNeighbours Regressor
kn=KNeighborsRegressor(n_neighbors=5,weights='distance')
kn.fit(X_train_scaled,y_train)

# Predict the missing values
y_pred=kn.predict(X_test_scaled)
print("KNeighbours MAE :", mean_absolute_error(y_test, y_pred))
print("KNeighbours MSE :", mean_squared_error(y_test, y_pred))
print("KNeighbours R2 :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = kn.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:', r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
KNeighbours MAE : 0.10688240425191081
KNeighbours MSE : 0.9335427549049045
KNeighbours R2 : 0.775552286508246
Training R2: 0.9999999980894768
Training MAE: 7.501875468867223e-07
Training MSE: 3.750937734433616e-09
```

In [67]:
```
# Training the model using Random Forest Regressor
rf_model =RandomForestRegressor(n_estimators=100,random_state=42)
rf_model.fit(X_train_scaled, y_train)
# Predict the missing values
y_pred = rf_model.predict(X_test_scaled)

print("Random Forest MAE :", mean_absolute_error(y_test, y_pred))
print("Random Forest MSE :", mean_squared_error(y_test, y_pred))
print("Random Forest R2 :",r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = rf_model.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:', r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
Random Forest MAE : 0.04133078057805784
Random Forest MSE : 0.7136476655050725
Random Forest R2 : 0.8284207274709576
Training R2: 0.9962532867526279
Training MAE: 0.010210304451112817
Training MSE: 0.007355937006199325
```

for the fature selction, the values with the highest correlation from the heatmap for numerical varibles and categorical variables.('NA_Sales','EU_Sales','JP_Sales', 'Other_Sales','Critic_Count','User_Count') were selected for numerical varibles while('Platform','Genre', 'Developer') were selected for categorical variables
from the above error functions, The linear regression model performed best in the prediction of the Global sales with the Mean absolute error of approximately 0.003 and a mean square error of 2.73e-05 approximately. Since the training loss is less than the test loss then it's confirmed that there isn't a case of overfitting.

Effect of the user and critic score as well as their reviews on NA sales

In [217]:
```
#Feature selection-picking the variables that significantly contribute to the Global sales
#from the heatmap above it is evident that the NA sales,EU sales ,JP sales and other sales contributes significantly to the glob
X=df[['Critic_Score','Critic_Count', 'User_Score', 'User_Count']]
y=df['NA_Sales']
# Define the input and output features
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
#Normalising dataset
scale=StandardScaler()#initialising Normaliser
scale.fit(X_train)#fitting Normaliser to train data
X_train_scaled=scale.transform(X_train)
X_test_scaled=scale.transform(X_test)
```

In [218]:
```
# Train the model using Linear regression
lm = LinearRegression()
lm.fit(X_train_scaled, y_train)

# Predict the missing values
y_pred = lm.predict(X_test_scaled)
print("Linear Regression MAE :", mean_absolute_error(y_test, y_pred))
print("Linear Regression MSE :", mean_squared_error(y_test, y_pred))
print("Linear Regression R2 score :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = lm.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
Linear Regression MAE : 0.3038680226303471
Linear Regression MSE : 0.9630697736870417
Linear Regression R2 score : 0.07158584231927467
Training R2: 0.10837429893648354
Training MAE: 0.2864653038795573
Training MSE: 0.5082637968144674
```

In [220]:
```
interceptm=lm.intercept_
coefficientm=lm.coef_
interceptm
coefficientm
'Critic_Score','Critic_Count', 'User_Score', 'User_Count'
Linear_model='NA_sales={:.3f} + {:.3f}*CS + {:.3f}*CC + {:.3f}*US +{:.3f} *UC'.format(interceptm,coefficientm[0],coefficientm[1]
Linear_model
```

Out[220]:
```
'NA_sales=0.260 + 0.116*CS + 0.134*CC + -0.019*US +0.092 *UC'
```

In [205]:
```
#Using Lasso regressor
X=df[['Critic_Score','Critic_Count', 'User_Score', 'User_Count']]
y=df['NA_Sales']
# Define the input and output features
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
#Normalising dataset
scale=StandardScaler()#initialising Normaliser
scale.fit(X_train)#fitting Normaliser to train data
X_train_scaled=scale.transform(X_train)
X_test_scaled=scale.transform(X_test)

# Train the model using Linear regression
lasso = Lasso(alpha=0.1)
lasso.fit(X_train_scaled, y_train)

# Predict the missing values
y_pred = lasso.predict(X_test_scaled)
print("Lasso Regression MAE :", mean_absolute_error(y_test, y_pred))
print("Lasso Regression MSE :", mean_squared_error(y_test, y_pred))
print("Lasso Regression R2 score :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = lasso.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
Lasso Regression MAE : 0.29216904059347704
Lasso Regression MSE : 0.9871278412845954
Lasso Regression R2 score : 0.04839349304794649
Training R2: 0.07580016125862188
Training MAE: 0.27636368534168454
Training MSE: 0.5268324124054706
```

In [206]:
```
#determining the intercept and coefficient of the predictions using lasso
interceptm=lasso.intercept_
coefficientm=lasso.coef_
interceptm
coefficientm
'Critic_Score','Critic_Count', 'User_Score', 'User_Count'
model='NA_sales={:.3f} + {:.3f}*CS + {:.3f}*CC + {:.3f}*US +{:.3f} *UC'.format(interceptm,coefficientm[0],coefficientm[1],coeffi
model# printing the model for NA sales
```

Out[206]:
```
'NA_sales=0.260 + 0.048*CS + 0.072*CC + 0.000*US +0.031 *UC'
```

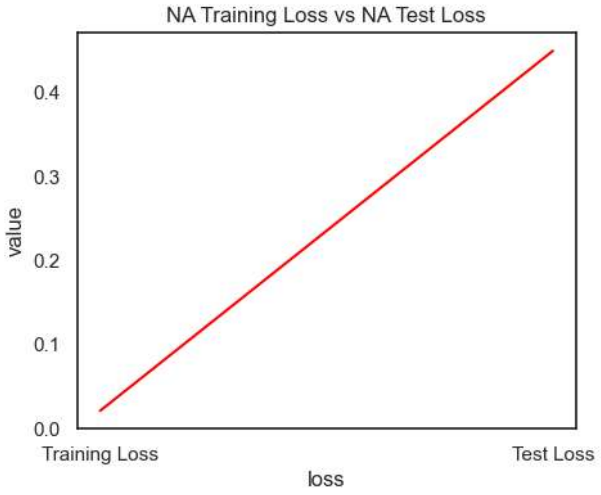
In [197]:
```
# Train the model using KNeighbours Regressor
kn=KNeighborsRegressor(n_neighbors=5,weights='distance')
kn.fit(X_train_scaled,y_train)

# Predict the missing values
y_pred=kn.predict(X_test_scaled)

print("KNeighbours MAE :", mean_absolute_error(y_test, y_pred))
print("KNeighbours MSE :", mean_squared_error(y_test, y_pred))
print("KNeighbours R2 score :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = kn.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
KNeighbours MAE : 0.23274862377489977
KNeighbours MSE : 0.9750950807157053
KNeighbours R2 score : 0.05999326034764185
Training R2: 0.999861949259382
Training MAE: 0.0002580645161290324
Training MSE: 7.86946736684171e-05
```

In [198]:
```
# Training the model using Random Forest Regressor
rf_model =RandomForestRegressor(n_estimators=100,random_state=42)
rf_model.fit(X_train_scaled, y_train)
# Predict the missing values
y_pred = rf_model.predict(X_test_scaled)

print("Random Forest MAE :", mean_absolute_error(y_test, y_pred))
print("Random Forest MSE :", mean_squared_error(y_test, y_pred))
print("Random forest R2 score :", r2_score(y_test, y_pred))

# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = rf_model.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
Random Forest MAE : 0.2324937498749875
Random Forest MSE : 0.9326166188068847
Random forest R2 score : 0.10094315464415426
Training R2: 0.9034112845150561
Training MAE: 0.07976014651877254
Training MSE: 0.055059591937797546
```

In [211]:
```
df_NA = pd.DataFrame({'Loss':['Training Loss', 'Test Loss'], 'Value':[mse, mean_squared_error(y_test, y_pred)]})
fig, ax = plt.subplots(figsize=(5, 4))
# Create the plot
sns.set(style="white")
sns.lineplot(x='Loss', y='Value', data=df_JP,color='red')
plt.xlabel('loss')
plt.ylabel('value')
plt.title('NA Training Loss vs NA Test Loss');
```



Effect of the user and critic score as well as their reviews on EU Sales

In [221]:
```
#Feature selection-picking the variables that significantly contribute to the Global sales
#from the heatmap above it is evident that the NA sales,EU sales ,JP sales and other sales contributes significantly to the glob
X=df[['Critic_Score','Critic_Count', 'User_Score', 'User_Count']]
y=df['EU_Sales']
# Define the input and output features
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
#Normalising dataset
scale=StandardScaler()#initialising Normaliser
scale.fit(X_train)#fitting Normaliser to train data
X_train_scaled=scale.transform(X_train)
X_test_scaled=scale.transform(X_test)
```

In [222]:
```
# Train the model using Linear regression
lm = LinearRegression()
lm.fit(X_train_scaled, y_train)

# Predict the missing values
y_pred = lm.predict(X_test_scaled)
print("Linear Regression MAE :", mean_absolute_error(y_test, y_pred))
print("Linear Regression MSE :", mean_squared_error(y_test, y_pred))
print("Linear Regression R2 score :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = lm.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
Linear Regression MAE : 0.18473706450101518
Linear Regression MSE : 0.4515410168557685
Linear Regression R2 score : 0.07359307510133772
Training R2: 0.14944616600155236
Training MAE: 0.1677861783551224
Training MSE: 0.16637340638119605
```

In [223]:
```
interceptm=lm.intercept_
coefficientm=lm.coef_
interceptm
coefficientm
'Critic_Score','Critic_Count', 'User_Score', 'User_Count'
Linear_model='EU_sales={:.3f} + {:.3f}*CS + {:.3f}*CC + {:.3f}*US +{:.3f} *UC'.format(interceptm,coefficientm[0],coefficientm[1]
Linear_model
```

Out[223]:
```
'EU_sales=0.142 + 0.060*CS + 0.105*CC + -0.022*US +0.071 *UC'
```

In [208]:
```
#Using Lasso regressor
X=df[['Critic_Score','Critic_Count', 'User_Score', 'User_Count']]
y=df['EU_Sales']
# Define the input and output features
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
#Normalising dataset
scale=StandardScaler()#initialising Normaliser
scale.fit(X_train)#fitting Normaliser to train data
X_train_scaled=scale.transform(X_train)
X_test_scaled=scale.transform(X_test)

# Train the model using Linear regression
lasso = Lasso(alpha=0.1)
lasso.fit(X_train_scaled, y_train)

# Predict the missing values
y_pred = lasso.predict(X_test_scaled)
print("Lasso Regression MAE :", mean_absolute_error(y_test, y_pred))
print("Lasso Regression MSE :", mean_squared_error(y_test, y_pred))
print("Lasso Regression R2 score :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = lasso.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
Lasso Regression MAE : 0.18738092326793615
Lasso Regression MSE : 0.4734899470484768
Lasso Regression R2 score : 0.028561416479863544
Training R2: 0.05853974959689168
Training MAE: 0.1724359665938978
Training MSE: 0.18415524399640154
```

In [210]:
```
interceptm=lasso.intercept_
coefficientm=lasso.coef_
interceptm
coefficientm
'Critic_Score','Critic_Count', 'User_Score', 'User_Count'
model='EU_sales={:.3f} + {:.3f}*CS + {:.3f}*CC + {:.3f}*US +{:.3f} *UC'.format(interceptm,coefficientm[0],coefficientm[1],coeffi
model
```

Out[210]:
```
'EU_sales=0.142 + 0.000*CS + 0.040*CC + 0.000*US +0.008 *UC'
```

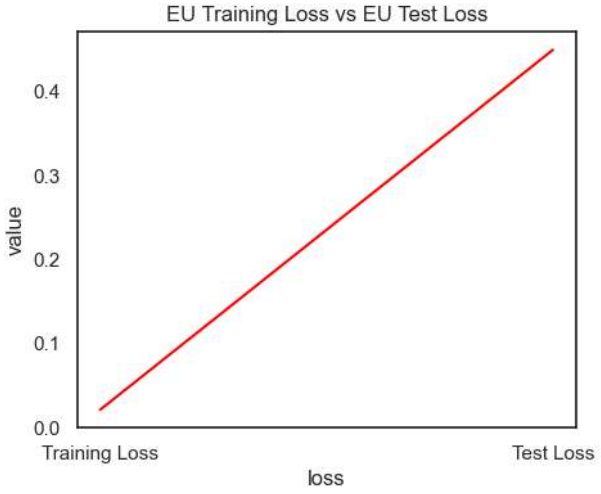
In [133]:
```
# Train the model using KNeighbours Regressor
kn=KNeighborsRegressor(n_neighbors=5,weights='distance')
kn.fit(X_train_scaled,y_train)

# Predict the missing values
y_pred=kn.predict(X_test_scaled)
print("KNeighbours MAE :", mean_absolute_error(y_test, y_pred))
print("KNeighbours MSE :", mean_squared_error(y_test, y_pred))
print("KNeighbours R2 score :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = kn.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
KNeighbours MAE : 0.15218344154270147
KNeighbours MSE : 0.4723768327145608
KNeighbours R2 score : 0.030845144399697433
Training R2: 0.9998013559386376
Training MAE: 0.0001732933233308327
Training MSE: 3.885596399099775e-05
```

In [134]:
```
# Training the model using Random Forest Regressor
rf_model =RandomForestRegressor(n_estimators=100,random_state=42)
rf_model.fit(X_train_scaled, y_train)
# Predict the missing values
y_pred = rf_model.predict(X_test_scaled)

print("Random Forest MAE :", mean_absolute_error(y_test, y_pred))
print("Random Forest MSE :", mean_squared_error(y_test, y_pred))
print("Random forest R2 score :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = rf_model.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
Random Forest MAE : 0.15140291100538625
Random Forest MSE : 0.4497714075033522
Random forest R2 score : 0.07722370509340226
Training R2: 0.892661839091527
Training MAE: 0.05029431361411781
Training MSE: 0.02099598491148321
```

In [212]:
```
df_EU = pd.DataFrame({'Loss':['Training Loss', 'Test Loss'], 'Value':[mse, mean_squared_error(y_test, y_pred)]})
fig, ax = plt.subplots(figsize=(5, 4))
# Create the plot
sns.set(style="white")
sns.lineplot(x='Loss', y='Value', data=df_JP,color='red')
plt.xlabel('loss')
plt.ylabel('value')
plt.title('EU Training Loss vs EU Test Loss');
```



Random forest gave a better Mean absolute error and Mean squared error(~0.15 and ~0.45) when compared to other estimators. There's a high probability that a higher number of critics and users as well as positive review scores would be associated with higher sales of video games in Europe.

Effect of the user and critic score as well as their reviews on JP sales

In [226]:
```
#Feature selection-picking the variables that significantly contribute to the Global sales
#from the heatmap above it is evident that the NA sales,EU sales ,JP sales and other sales contributes significantly to the glob
X=df[['Critic_Score','Critic_Count', 'User_Score', 'User_Count']]
y=df['JP_Sales']
# Define the input and output features
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
#Normalising dataset
scale=StandardScaler()#initialising Normaliser
scale.fit(X_train)#fitting Normaliser to train data
X_train_scaled=scale.transform(X_train)
X_test_scaled=scale.transform(X_test)
```

In [227]:
```
# Train the model using Linear regression
lm = LinearRegression()
lm.fit(X_train_scaled, y_train)

# Predict the missing values
y_pred = lm.predict(X_test_scaled)
print("Linear Regression MAE :", mean_absolute_error(y_test, y_pred))
print("Linear Regression MSE :", mean_squared_error(y_test, y_pred))
print("Linear Regression R2 score :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = lm.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
Linear Regression MAE : 0.1149292908614344
Linear Regression MSE : 0.09015092765205976
Linear Regression R2 score : 0.06642682057144667
Training R2: 0.06291880613825418
Training MAE: 0.11015275967600013
Training MSE: 0.08943209859020083
```

In [228]:
```
interceptm=lm.intercept_
coefficientm=lm.coef_
interceptm
coefficientm
'Critic_Score','Critic_Count', 'User_Score', 'User_Count'
Linear_model='JP_sales={:.3f} + {:.3f}*CS + {:.3f}*CC + {:.3f}*US +{:.3f} *UC'.format(interceptm,coefficientm[0],coefficientm[1]
model
```

Out[228]:
```
'JP_sales=0.07716 + 0.00000*CS + 0.00000*CC + 0.00000*US +0.00000 *UC'
```

In [214]:
```
#Using Lasso regressor
X=df[['Critic_Score','Critic_Count', 'User_Score', 'User_Count']]
y=df['JP_Sales']
# Define the input and output features
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state=42)
#Normalising dataset
scale=StandardScaler()#initialising Normaliser
scale.fit(X_train)#fitting Normaliser to train data
X_train_scaled=scale.transform(X_train)
X_test_scaled=scale.transform(X_test)

# Train the model using Linear regression
lasso = Lasso(alpha=0.1)
lasso.fit(X_train_scaled, y_train)

# Predict the missing values
y_pred = lasso.predict(X_test_scaled)
print("Lasso Regression MAE :", mean_absolute_error(y_test, y_pred))
print("Lasso Regression MSE :", mean_squared_error(y_test, y_pred))
print("Lasso Regression R2 score :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = lasso.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
Lasso Regression MAE : 0.11792638506161195
Lasso Regression MSE : 0.09657590378734608
Lasso Regression R2 score : -0.00010810651795623016
Training R2: 0.0
Training MAE: 0.11438909262199269
Training MSE: 0.09543687268084837
```

In [216]:
```
interceptm=lasso.intercept_
coefficientm=lasso.coef_
interceptm
coefficientm
'Critic_Score','Critic_Count', 'User_Score', 'User_Count'
model='JP_sales={:.5f} + {:.5f}*CS + {:.5f}*CC + {:.5f}*US +{:.5f} *UC'.format(interceptm,coefficientm[0],coefficientm[1],coeffi
model
```

Out[216]:
```
'JP_sales=0.07716 + 0.00000*CS + 0.00000*CC + 0.00000*US +0.00000 *UC'
```

In [80]:
```
# Train the model using KNeighbours Regressor
kn=KNeighborsRegressor(n_neighbors=5,weights='distance')
kn.fit(X_train_scaled,y_train)

# Predict the missing values
y_pred=kn.predict(X_test_scaled)
print("KNeighbours MAE :", mean_absolute_error(y_test, y_pred))
print("KNeighbours MSE :", mean_squared_error(y_test, y_pred))
print("Kneighbours R2 score :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = kn.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
KNeighbours MAE : 0.07642766795974257
KNeighbours MSE : 0.08515891162280907
Kneighbours R2 score : 0.11812248691192939
Training R2: 0.9999764969167347
Training MAE: 2.550637659414863e-05
Training MSE: 2.243060765191298e-06
```

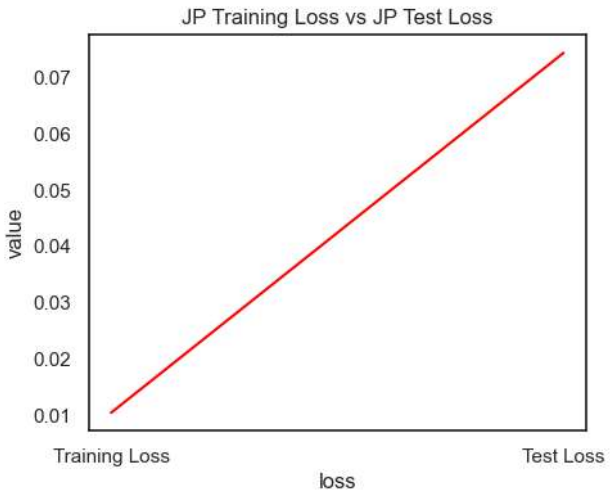
In [81]:
```
# Training the model using Random Forest Regressor
rf_model =RandomForestRegressor(n_estimators=100,random_state=42)
rf_model.fit(X_train_scaled, y_train)
# Predict the missing values
y_pred = rf_model.predict(X_test_scaled)

print("Random Forest MAE :", mean_absolute_error(y_test, y_pred))
print("Random Forest MSE :", mean_squared_error(y_test, y_pred))
print("Random forest R2 score :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = rf_model.predict(X_train_scaled)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
Random Forest MAE : 0.07697395903876102
Random Forest MSE : 0.07440936373302444
Random forest R2 score : 0.22944124826309376
Training R2: 0.8912629445080988
Training MAE: 0.027883582270567645
Training MSE: 0.010377524520670917
```

In [82]:
```
# Training the model using Random Forest Regressor without scaling
rf_model =RandomForestRegressor(n_estimators=100,random_state=42)
rf_model.fit(X_train, y_train)
# Predict the missing values
y_pred = rf_model.predict(X_test)

print("Random Forest MAE :", mean_absolute_error(y_test, y_pred))
print("Random Forest MSE :", mean_squared_error(y_test, y_pred))
print("Random forest R2 score :", r2_score(y_test, y_pred))
# Calculate the mean abs error and Mean Sqr error on the training set
y_train_pred = rf_model.predict(X_train)
mae=mean_absolute_error(y_train, y_train_pred)
mse = mean_squared_error(y_train, y_train_pred)
r2=r2_score(y_train, y_train_pred)
print('Training R2:',r2)
print('Training MAE:', mae)
print('Training MSE:', mse)
```

```
Random Forest MAE : 0.07688306494935207
Random Forest MSE : 0.07442947182098325
Random forest R2 score : 0.22923301555715436
Training R2: 0.8913267985017392
Training MAE: 0.027898323330832715
Training MSE: 0.010371430495209699
```

In [83]:
```
# Create a DataFrame to store the training and test loss values
df_JP = pd.DataFrame({'Loss':['Training Loss', 'Test Loss'], 'Value':[mse, mean_squared_error(y_test, y_pred)]})
fig, ax = plt.subplots(figsize=(5, 4))
# Create the plot
sns.set(style="white")
sns.lineplot(x='Loss', y='Value', data=df_JP,color='red')
plt.xlabel('loss')
plt.ylabel('value')
plt.title('JP Training Loss vs JP Test Loss');
```



The choice of regressor was based on the model performance on the training and test data. For the NA sales, the random forest regressor mean absolute errors and Mean square errors were lesser(0.23 and 0.93 respectively) than the results from the KNeighbours regressor and Linear Regressor.

Using all relevant categorical columns as target

In [229]:
```
relevant_columns=[ 'Platform', 'Genre', 'Rating']# based on the number of unique values
```

In [239]:
```
#Using KNeighbours classifier
# Loop through each categorical column
print('Using KNeighbours Classifier \n')
for target_col in relevant_columns:
    # Split into X (features) and y (target)
    X = df.drop(columns=[target_col])
    y = df[target_col]

    # Encode categorical target variable
    le = LabelEncoder()
    y = le.fit_transform(y)

    # Split into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Normalising dataset
    scale = StandardScaler() #initialising Normaliser
    scale.fit(X_train) #fitting Normaliser to train data
    X_train_scaled = scale.transform(X_train)
    X_test_scaled = scale.transform(X_test)
    # Instatiating and fitting the model
    kn_clf = KNeighborsClassifier(n_neighbors=5, weights='distance', algorithm='auto')
    kn_clf.fit(X_train_scaled, y_train)

    # Predict the missing values
    y_pred = kn_clf.predict(X_test_scaled)

    # Decode predicted values
    y_pred = le.inverse_transform(y_pred)

    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy score for {target_col}: {accuracy:.5f}")
```

```
Using KNeighbours Classifier

Accuracy score for Platform: 0.35374
Accuracy score for Genre: 0.31983
Accuracy score for Rating: 0.58356
```

In [86]:
```
#Using Logistic Regression
logreg_clf=LogisticRegression()
grid_param= {
    'penalty': ['l1','l2'],
    'solver':['newton-cg','lbfgs','liblinear']
}

gcsv=GridSearchCV(logreg_clf,grid_param,scoring='accuracy',cv=5)
gcsv.fit(X_train_scaled, y_train)
```

Out[86]:
```
GridSearchCV(cv=5, estimator=LogisticRegression(),
             param_grid={'penalty': ['l1', 'l2'],
                         'solver': ['newton-cg', 'lbfgs', 'liblinear']},
             scoring='accuracy')
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [87]:
```
print("Tuned Hyperparameters :",gcsv.best_params_)
print("Accuracy :",gcsv.best_score_)
```

```
Tuned Hyperparameters : {'penalty': 'l2', 'solver': 'lbfgs'}
Accuracy : 0.5228057014253563
```

In [233]:
```
print('Using Logistic Regression \n')
for target_col in relevant_columns:
    # Instatiating and fitting the model
    #Using the best parameters {'penalty': 'l1', 'solver': 'liblinear'}
    logreg_clf=LogisticRegression(penalty='l2',solver='liblinear')
    logreg_clf.fit(X_train_scaled, y_train)

    # Predict the missing values
    y_pred=logreg_clf.predict(X_test_scaled)

    # Decode predicted values
    y_pred = le.inverse_transform(y_pred)

    accuracy = accuracy_score(y_test, y_pred)

    print(f"Accuracy score for {target_col}: {accuracy:.5f}")
```

```
Using Logistic Regression

Accuracy score for Platform: 0.51845
Accuracy score for Genre: 0.51845
Accuracy score for Rating: 0.51845
```

In [234]:
```
print('Using Decision Tree Classifier \n')
for target_col in relevant_columns:
    DT_clf=DecisionTreeClassifier()
    DT_clf.fit(X_train_scaled, y_train)

    # Predict the missing values
    y_pred=DT_clf.predict(X_test_scaled)

    # Decode predicted values
    y_pred = le.inverse_transform(y_pred)

    accuracy = accuracy_score(y_test, y_pred)

    print(f"Accuracy score for {target_col}: {accuracy:.5f}")
```

```
Using Decision Tree Classifier

Accuracy score for Platform: 0.66247
Accuracy score for Genre: 0.66277
Accuracy score for Rating: 0.66037
```

In [90]:
```
print('Using Random Forest Classifier \n')
for target_col in relevant_columns:
    # Instatiating and fitting the model

    rf_clf=RandomForestClassifier()
    rf_clf.fit(X_train_scaled, y_train)

    # Predict the missing values
    y_pred=rf_clf.predict(X_test_scaled)

    # Decode predicted values
    y_pred = le.inverse_transform(y_pred)
    # predicting accuracy on training and test data
    accuracy = accuracy_score(y_test, y_pred)

    print(f"Accuracy score for {target_col}: {accuracy:.5f}")
```

```
Using Random Forest Classifier

Accuracy score for Platform: 0.73807
Accuracy score for Genre: 0.73417
Accuracy score for Rating: 0.73447
```

From the predictions above, the Random Forest classifier achieved the best output and it can be seen from the results that rating best classifies the dataset with an accuracy of 74% percent. when compared to the accuracy for Genre at 73.62% and platform at 73.57%.

Testing for overfitting using the kfold cross validation set

The results for a 5 fold cross validation set shows some variances across various subsets of the data. however, a mean score of 0.42 and standard deviation of 0.044 suggests a relatively moderate performance. this further reveals that there will be need for hyper parameter tuning to achieve a more stable result.

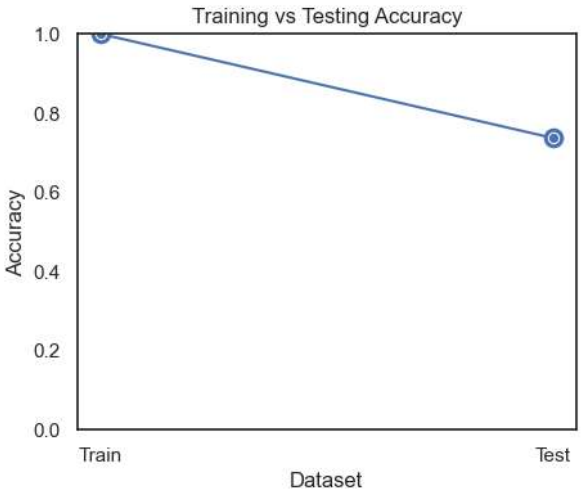
In [255]:
```
print('Using Random Forest Classifier \n')
for target_col in relevant_columns:
    # Instatiating and fitting the model

    rf_clf=RandomForestClassifier()
    rf_clf.fit(X_train_scaled, y_train)

    # Predict the target
    y_pred_train = kn_clf.predict(X_train_scaled)
    y_pred=rf_clf.predict(X_test_scaled)

    # Decode predicted values
    y_pred = le.inverse_transform(y_pred)
    y_pred_train = le.inverse_transform(y_pred_train)
    # predicting accuracy on training and test data
    accuracy = accuracy_score(y_test, y_pred)

    print(f"Accuracy score for {target_col}: {accuracy:.5f}")
# Calculate accuracy scores
    accuracy_train = accuracy_score(y_train, y_pred_train)
    accuracy_test = accuracy_score(y_test, y_pred)

    print(f"Accuracy score for {target_col} (Training Set): {accuracy_train:.5f}")
    print(f"Accuracy score for {target_col} (Test Set): {accuracy_test:.5f}")
```

```
Using Random Forest Classifier

Accuracy score for Platform: 0.73717
Accuracy score for Platform (Training Set): 1.00000
Accuracy score for Platform (Test Set): 0.73717
Accuracy score for Genre: 0.73837
Accuracy score for Genre (Training Set): 1.00000
Accuracy score for Genre (Test Set): 0.73837
Accuracy score for Rating: 0.73687
Accuracy score for Rating (Training Set): 1.00000
Accuracy score for Rating (Test Set): 0.73687
```

In [244]:
```
#without using the scaled data and using the default parameters
print('Using Random Forest Classifier \n')
for target_col in relevant_columns:
    # Instatiating and fitting the model

    rf_clf=RandomForestClassifier()
    rf_clf.fit(X_train, y_train)

    # Predict the missing values
    y_pred=rf_clf.predict(X_test)

    # Decode predicted values
    y_pred = le.inverse_transform(y_pred)
    # predicting accuracy on training and test data
    accuracy_train = accuracy_score(y_train, y_pred_train)
    accuracy = accuracy_score(y_test, y_pred)

    print(f"Accuracy score for {target_col} (Training Set): {accuracy_train:.5f}")
    print(f"Accuracy score for {target_col}: {accuracy:.5f}")
    print(" Classification Report \n")
    print(classification_report(y_test,y_pred))
```

```
Using Random Forest Classifier

Accuracy score for Platform (Training Set): 1.00000
Accuracy score for Platform: 0.73297
 Classification Report

              precision    recall  f1-score   support

           1       0.71      0.85      0.78      1173
           2       0.76      0.44      0.56       432
           3       0.00      0.00      0.00         3
           5       0.67      0.60      0.64       405
           6       0.00      0.00      0.00         1
           7       0.76      0.77      0.77      1319

    accuracy                           0.73      3333
   macro avg       0.49      0.44      0.46      3333
weighted avg       0.73      0.73      0.73      3333

Accuracy score for Genre (Training Set): 1.00000
Accuracy score for Genre: 0.73327
 Classification Report

              precision    recall  f1-score   support

           1       0.71      0.86      0.78      1173
           2       0.79      0.43      0.56       432
           3       0.00      0.00      0.00         3
           5       0.67      0.61      0.64       405
           6       0.00      0.00      0.00         1
           7       0.76      0.76      0.76      1319

    accuracy                           0.73      3333
   macro avg       0.49      0.44      0.46      3333
weighted avg       0.74      0.73      0.73      3333

Accuracy score for Rating (Training Set): 1.00000
Accuracy score for Rating: 0.73087
 Classification Report

              precision    recall  f1-score   support

           1       0.71      0.85      0.77      1173
           2       0.78      0.43      0.55       432
           3       0.00      0.00      0.00         3
           5       0.67      0.59      0.63       405
           6       0.00      0.00      0.00         1
           7       0.76      0.77      0.76      1319

    accuracy                           0.73      3333
   macro avg       0.49      0.44      0.45      3333
weighted avg       0.73      0.73      0.72      3333
```

In [266]:
```
accuracy_df = pd.DataFrame({
    'accuracy': [accuracy_train, accuracy],
    'dataset': ['Train', 'Test']
})
fig, ax = plt.subplots(figsize=(5, 4))
sns.lineplot(x='dataset', y='accuracy', data=accuracy_df, marker='o')
sns.scatterplot(x='dataset', y='accuracy', data=accuracy_df, marker='o', s=150)
plt.title('Training vs Testing Accuracy')
plt.xlabel('Dataset')
plt.ylabel('Accuracy')
plt.ylim(0, 1)
plt.show()
fig.savefig("Training and Testing Accuracy.png", dpi=300)
```



without using the scaled data and using the default parameters, there was a slight drop in the accuracy score from 73.9% to 73.8% and the performance of the model revealed that the developer would best clasify the dataset with an accuracy score of 73.8%

In [240]:
```
Cross_val=StratifiedKFold(n_splits=6,shuffle=True,random_state=42)
scores=cross_val_score(rf_clf,X,y,cv=5)
print('Cross Validation scores :', scores)
print('Mean Score :', scores.mean())
print('Standard deviation Score :', scores.std())
print(f"Accuracy score : {accuracy:.5f}")
```

```
Cross Validation scores : [0.35013501 0.47164716 0.4320432  0.42617047 0.41866747]
Mean Score : 0.41973266354246475
Standard deviation Score : 0.03932703322139715
Accuracy score : 0.58356
```

Hyper Parameter tuning-Optimizing the estimator/ parameters to improve the performance of the model

In [93]:
```
print(rf_clf.get_params().keys())
```

```
dict_keys(['bootstrap', 'ccp_alpha', 'class_weight', 'criterion', 'max_depth', 'max_features', 'max_leaf_nodes', 'max_samples',
'min_impurity_decrease', 'min_samples_leaf', 'min_samples_split', 'min_weight_fraction_leaf', 'n_estimators', 'n_jobs', 'oob_sc
ore', 'random_state', 'verbose', 'warm_start'])
```

In [94]:
```
grid_param= {
    'n_estimators': [10,100,200],
    'max_features':["auto", "sqrt", "log2"],
}

gcsv=GridSearchCV(rf_clf,grid_param,scoring='accuracy',cv=5)
gcsv.fit(X_train, y_train)
```

Out[94]:
```
GridSearchCV(cv=5, estimator=RandomForestClassifier(),
             param_grid={'max_features': ['auto', 'sqrt', 'log2'],
                         'n_estimators': [10, 100, 200]},
             scoring='accuracy')
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [95]:
```
print("Tuned Hyperparameters :",gcsv.best_params_)
print("Accuracy :",gcsv.best_score_)
```

```
Tuned Hyperparameters : {'max_features': 'auto', 'n_estimators': 200}
Accuracy : 0.7301575393848463
```

Using the hyperparameters suggested above doesn't improve the accuracy of the model as the accuracy dropped from 74% to 73.1%; the model can be deployed in practice as the max accuracy of the model improved from 58% to 73.4%

Clustering problem

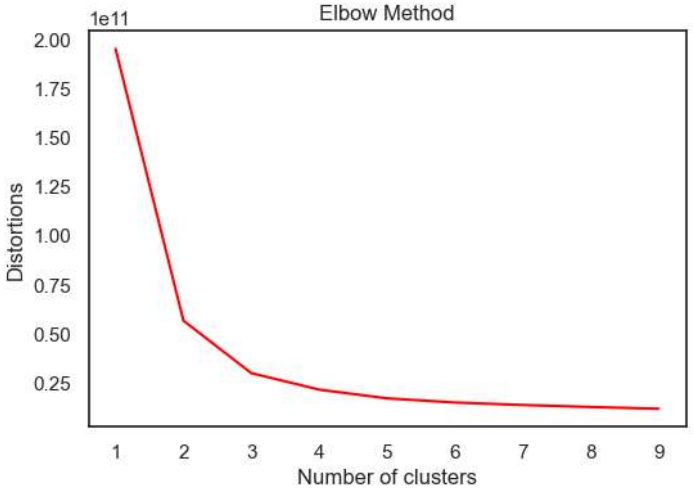
In [267]:
```
X = df.drop(columns=[target_col])
y = df[target_col]
relevant_columns=['Genre','Rating','Platform'] #columns with small unique values and accuracy for Genre and Rating was 73.4 and
```

In [268]:
```
KMeans()
```

Out[268]:
```
KMeans()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [281]:
```
distortion =[KMeans(n_clusters=k, random_state=42).fit(X).inertia_ for k in range(1, 10)]# using list comprehension
fig, ax = plt.subplots(figsize=(6, 4))
sns.lineplot(x=range(1, 10),y=distortion,color='red')
plt.ylabel('Distortions')
plt.xlabel('Number of clusters')
plt.title('Elbow Method');
fig.savefig("Number of clusters elbow method.png", dpi=300)
```



In [99]:
```
#Using KMeans as the clustering Algorithm
for target_col in relevant_columns:
    # Split into X (features) and y (target)
    X = df.drop(columns=[target_col])
    y_true = df[target_col]
    scaler=StandardScaler()
    scaled_df=scaler.fit_transform(X)
    kmeans = KMeans(n_clusters=3, random_state=42)# Number of clusters from elbow method
    C_labels = kmeans.fit_predict(scaled_df)
    # Calculate external evaluation measures
    v_measure = v_measure_score(y_true, C_labels)
    rand_index = adjusted_rand_score(y_true, C_labels)
    mi_score = adjusted_mutual_info_score(y_true, C_labels)

    print(f'External evaluation measures')
    print(f'****************************************')
    print(f'V-measure score for {target_col}: {v_measure:.3f}')
    print(f'Rand index score for {target_col}: {rand_index:.3f}')
    print(f'Mutual information score for {target_col}: {mi_score:.3f}\n')

    # Calculate internal evaluation measures
    db_score = davies_bouldin_score(scaled_df, C_labels)
    sil_score = silhouette_score(scaled_df, C_labels)
    cal_score = calinski_harabasz_score(scaled_df, C_labels)

    print(f'Internal evaluation measures')
    print(f'****************************************')
    print(f'Davies Bouldin Index for {target_col}: {db_score:.3f}')
    print(f'Silhouette Score for {target_col}: {sil_score:.3f}')
    print(f'Calinski Harabasz Score for {target_col}: {cal_score:.3f}\n')
```

```
External evaluation measures
****************************************
V-measure score for Genre: 0.014
Rand index score for Genre: 0.009
Mutual information score for Genre: 0.013

Internal evaluation measures
****************************************
Davies Bouldin Index for Genre: 2.042
Silhouette Score for Genre: 0.149
Calinski Harabasz Score for Genre: 2277.560

External evaluation measures
****************************************
V-measure score for Rating: 0.016
Rand index score for Rating: 0.002
Mutual information score for Rating: 0.016

Internal evaluation measures
****************************************
Davies Bouldin Index for Rating: 2.041
Silhouette Score for Rating: 0.144
Calinski Harabasz Score for Rating: 2286.396

External evaluation measures
****************************************
V-measure score for Platform: 0.062
Rand index score for Platform: 0.021
Mutual information score for Platform: 0.061

Internal evaluation measures
****************************************
Davies Bouldin Index for Platform: 1.994
Silhouette Score for Platform: 0.159
Calinski Harabasz Score for Platform: 2290.279
```

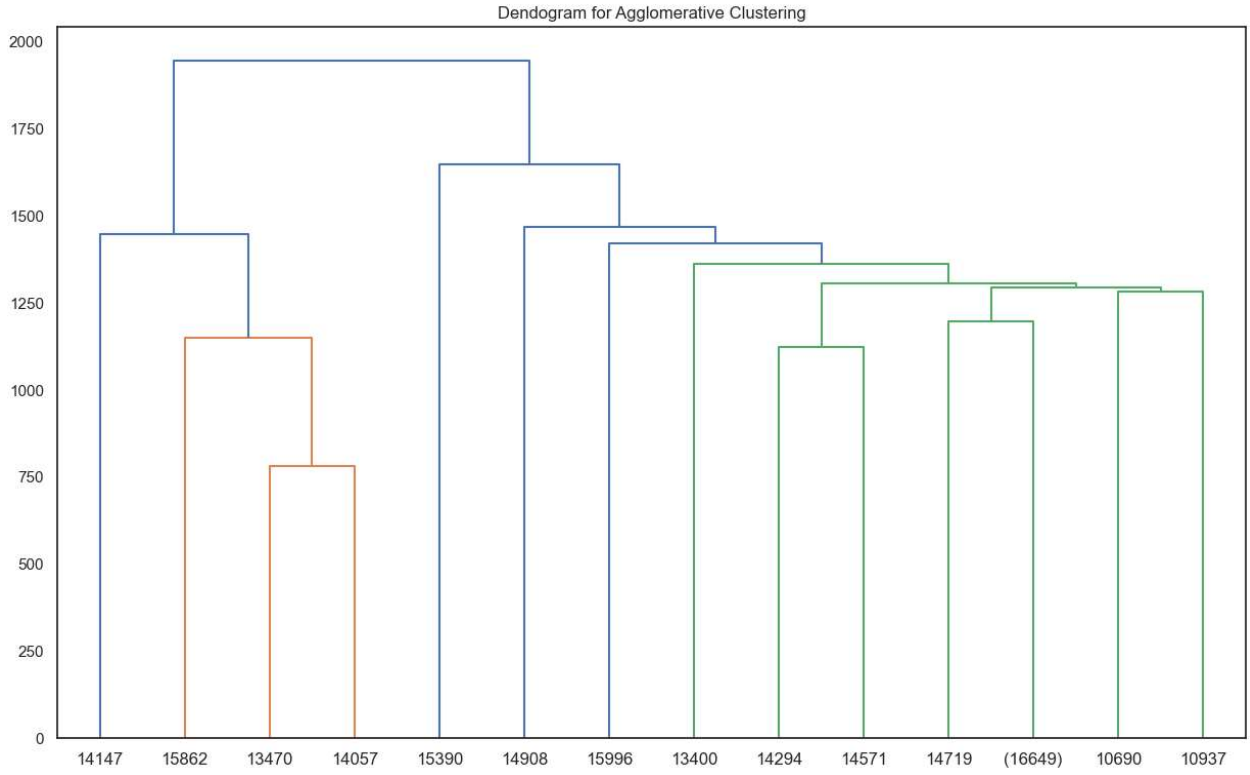
In [100]:
```python
#Using DBSCAN as the clustering algorithm
for target_col in relevant_columns:
    # Create ground truth labels
    y_true = df[target_col]

    # Drop the ground truth column from the features
    X = df.drop(columns=[target_col])

    # Perform scaling
    scaler = StandardScaler()
    scaled_df = scaler.fit_transform(X)

    # Cluster using DBSCAN Clustering
    dbscan=DBSCAN(eps=0.5,min_samples=5) #using 2 clusters
    C_labels = dbscan.fit_predict(scaled_df)

    # Calculate external evaluation measures
    v_measure = v_measure_score(y_true, C_labels)
    rand_index = adjusted_rand_score(y_true, C_labels)
    mi_score = adjusted_mutual_info_score(y_true, C_labels)

    print(f'External evaluation measures')
    print(f'**************************************')
    print(f'V-measure score for {target_col}: {v_measure:.5f}')
    print(f'Rand index score for {target_col}: {rand_index:.5f}')
    print(f'Mutual information score for {target_col}: {mi_score:.5f}\n')

    # Calculate internal evaluation measures
    db_score = davies_bouldin_score(scaled_df, C_labels)
    sil_score = silhouette_score(scaled_df, C_labels)
    cal_score = calinski_harabasz_score(scaled_df, C_labels)

    print(f'Internal evaluation measures')
    print(f'**************************************')
    print(f'Davies Bouldin Index for {target_col}: {db_score:.5f}')
    print(f'Silhouette Score for {target_col}: {sil_score:.5f}')
    print(f'Calinski Harabasz Score for {target_col}: {cal_score:.5f}\n')
```

```
External evaluation measures
**************************************
V-measure score for Genre: 0.03600
Rand index score for Genre: -0.00082
Mutual information score for Genre: 0.01790

Internal evaluation measures
**************************************
Davies Bouldin Index for Genre: 1.49299
Silhouette Score for Genre: -0.46149
Calinski Harabasz Score for Genre: 4.93036

External evaluation measures
**************************************
V-measure score for Rating: 0.02092
Rand index score for Rating: -0.00238
Mutual information score for Rating: 0.01494

Internal evaluation measures
**************************************
Davies Bouldin Index for Rating: 1.61788
Silhouette Score for Rating: -0.44794
Calinski Harabasz Score for Rating: 3.24293

External evaluation measures
**************************************
V-measure score for Platform: 0.03837
Rand index score for Platform: 0.00029
Mutual information score for Platform: 0.02030

Internal evaluation measures
**************************************
Davies Bouldin Index for Platform: 1.55539
Silhouette Score for Platform: -0.45303
Calinski Harabasz Score for Platform: 4.14619
```

In [278]:
```
#Using Agglomerative clustering algorithm
for target_col in relevant_columns:
    # Create ground truth labels
    y_true = df[target_col]

    # Drop the ground truth column from the features
    X = df.drop(columns=[target_col])

    # Perform scaling
    scaler = StandardScaler()
    scaled_df = scaler.fit_transform(X)

    # Cluster using Agglomerative Clustering
    agg_model = AgglomerativeClustering(n_clusters=2, linkage='single')#using 2 clusters
    C_labels = agg_model.fit_predict(scaled_df)

    # Calculate external evaluation measures
    v_measure = v_measure_score(y_true, C_labels)
    rand_index = adjusted_rand_score(y_true, C_labels)
    mi_score = adjusted_mutual_info_score(y_true, C_labels)

    print(f'External evaluation measures')
    print(f'**************************************')
    print(f'V-measure score for {target_col}: {v_measure:.5f}')
    print(f'Rand index score for {target_col}: {rand_index:.5f}')
    print(f'Mutual information score for {target_col}: {mi_score:.5f}\n')

    # Calculate internal evaluation measures
    db_score = davies_bouldin_score(scaled_df, C_labels)
    sil_score = silhouette_score(scaled_df, C_labels)
    cal_score = calinski_harabasz_score(scaled_df, C_labels)

    print(f'Internal evaluation measures')
    print(f'**************************************')
    print(f'Davies Bouldin Index for {target_col}: {db_score:.5f}')
    print(f'Silhouette Score for {target_col}: {sil_score:.5f}')
    print(f'Calinski Harabasz Score for {target_col}: {cal_score:.5f}\n')
```

```
External evaluation measures
**************************************
V-measure score for Genre: 0.00010
Rand index score for Genre: -0.00001
Mutual information score for Genre: -0.00002

Internal evaluation measures
**************************************
Davies Bouldin Index for Genre: 0.03162
Silhouette Score for Genre: 0.95429
Calinski Harabasz Score for Genre: 751.28017

External evaluation measures
**************************************
V-measure score for Rating: 0.00010
Rand index score for Rating: -0.00001
Mutual information score for Rating: -0.00002

Internal evaluation measures
**************************************
Davies Bouldin Index for Rating: 0.03160
Silhouette Score for Rating: 0.95430
Calinski Harabasz Score for Rating: 751.32256

External evaluation measures
**************************************
V-measure score for Platform: 0.00011
Rand index score for Platform: -0.00000
Mutual information score for Platform: -0.00001

Internal evaluation measures
**************************************
Davies Bouldin Index for Platform: 0.03165
Silhouette Score for Platform: 0.95427
Calinski Harabasz Score for Platform: 751.30623
```

In [288]:
```
from scipy.cluster import hierarchy
from scipy.cluster.hierarchy import dendrogram
fig, ax = plt.subplots(figsize=(15, 9))
matrix_linkage=hierarchy.linkage(agg_model.children_)
dendrogram(Z=matrix_linkage,truncate_mode='level',p=7);
ax.set_title('Dendogram for Agglomerative Clustering')
fig.savefig("Dendogram Agglomerative clustering.png", dpi=300)
```



In [103]:
```
#Using Agglomerative clustering algorithm
for target_col in relevant_columns:
    # Create ground truth labels
    y_true = df[target_col]

    # Drop the ground truth column from the features
    X = df.drop(columns=[target_col])

    # Perform scaling
    scaler = StandardScaler()
    scaled_df = scaler.fit_transform(X)

    # Cluster using Agglomerative Clustering
    agg_model = AgglomerativeClustering(n_clusters=3, linkage='single') #using 3 clusters
    C_labels = agg_model.fit_predict(scaled_df)

    # Calculate external evaluation measures
    v_measure = v_measure_score(y_true, C_labels)
    rand_index = adjusted_rand_score(y_true, C_labels)
    mi_score = adjusted_mutual_info_score(y_true, C_labels)

    print(f'External evaluation measures')
    print(f'**************************************')
    print(f'V-measure score for {target_col}: {v_measure:.5f}')
    print(f'Rand index score for {target_col}: {rand_index:.5f}')
    print(f'Mutual information score for {target_col}: {mi_score:.5f}\n')

    # Calculate internal evaluation measures
    db_score = davies_bouldin_score(scaled_df, C_labels)
    sil_score = silhouette_score(scaled_df, C_labels)
    cal_score = calinski_harabasz_score(scaled_df, C_labels)

    print(f'Internal evaluation measures')
    print(f'**************************************')
    print(f'Davies Bouldin Index for {target_col}: {db_score:.5f}')
    print(f'Silhouette Score for {target_col}: {sil_score:.5f}')
    print(f'Calinski Harabasz Score for {target_col}: {cal_score:.5f}\n')
```

```
External evaluation measures
**************************************
V-measure score for Genre: 0.00024
Rand index score for Genre: -0.00002
Mutual information score for Genre: -0.00005

Internal evaluation measures
**************************************
Davies Bouldin Index for Genre: 0.21144
Silhouette Score for Genre: 0.90604
Calinski Harabasz Score for Genre: 561.67099

External evaluation measures
**************************************
V-measure score for Rating: 0.00026
Rand index score for Rating: 0.00003
Mutual information score for Rating: -0.00001

Internal evaluation measures
**************************************
Davies Bouldin Index for Rating: 0.21147
Silhouette Score for Rating: 0.90606
Calinski Harabasz Score for Rating: 561.70213

External evaluation measures
**************************************
V-measure score for Platform: 0.00028
Rand index score for Platform: -0.00003
Mutual information score for Platform: -0.00002

Internal evaluation measures
**************************************
Davies Bouldin Index for Platform: 0.21203
Silhouette Score for Platform: 0.90600
Calinski Harabasz Score for Platform: 561.70421
```

Agglomerative clustering achieved the best performance for the quality of the clustering when compared to the KMeans and DBSCAN;Using the Agglomerative clustering algorithm, performance for Davies bouldin index was 0.21144,0.21147 for rating and 0.21203 for platform,using the silouhette performance evaluation matrix, the score for Genre was 0.90604, for rating was 0.90606 and 0.90600 for platform. Although the performance evaluation indices were similar, Rating formed more cohesive and quality groups when compared to other relevant columns. Other algorithms performed didn't perform well as the kmeans was 0.2 and silhouette score was 0.9 onthe other hand, the davies bouldin index for the relevant columns using Kmeans was above 2 and above 1.5 for DBSCAN which shows the clusters were not properly formed. additionally, the silouhette scores for the relevant columns using kmeans was closer to zero and negative indices for DBSCAN

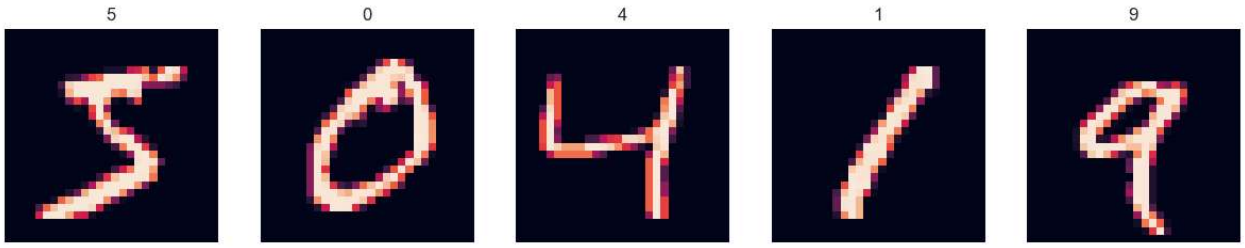
Component 3- Handwritten Recognition

In [378]:
```
#Data Exploration
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784')

print(mnist.data.shape)
# To check the description of the dataset
#print(mnist.DESCR)
```

```
(70000, 784)
```

In [300]:
```
#importing standard libraries
import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout
from tensorflow.keras.optimizers import SGD, Adam, RMSprop, Adagrad
```

In [301]:
```
# Load the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Define the class names (digits 0-9)
class_names = [str(i) for i in range(10)]

# Create a figure with 5 subplots
fig, axs = plt.subplots(1, 5, figsize=(15, 3))

# Loop through the first 5 images
for i in range(5):
    # Display the i-th image in the i-th subplot
    axs[i].imshow(X_train[i])

    # Set the title of the i-th subplot to the label of the i-th image
    axs[i].set_title(class_names[y_train[i]])

    # Turn off the axis labels and ticks for the i-th subplot
    axs[i].axis('off')

# Show the plot
plt.show()
fig.savefig("Handwritten digit classification.png", dpi=300)
```



In [302]:
```
# load train and test dataset
def load_dataset():
    # load dataset
    (X_train, y_train), (X_test, y_test) = mnist.load_data()
    # reshape dataset to have a single channel
    X_train = X_train.reshape((X_train.shape[0], 28, 28, 1)).astype('float32')
    X_test = X_test.reshape((X_test.shape[0], 28, 28, 1)).astype('float32')
    # Normalize and scale image pixels
    X_train = X_train / 255
    X_test = X_test / 255
    # one hot encode target values
    y_train = keras.utils.to_categorical(y_train).astype(int)
    y_test = keras.utils.to_categorical(y_test)
    return X_train, y_train, X_test, y_test
```

In [311]:
```
def Non_reg_model():#Model without regularisation
    model = Sequential()
    model.add(Conv2D(filters=32, kernel_size=(5, 5), input_shape=(28, 28, 1), activation='relu', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(filters=64, kernel_size=(5, 5), activation='relu', padding='same'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(10, activation='softmax')) #the possible different classes
    return model
```

In [305]:
```
# Building cnn model with 2 convolutional blocks and dropout of 0.2
def define_model():#function for 2 convolution blocks
    model = Sequential()
    model.add(Conv2D(filters=32, kernel_size=(5, 5), input_shape=(28, 28, 1), activation='relu', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(filters=64, kernel_size=(5, 5), activation='relu', padding='same'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(10, activation='softmax')) #the possible different classes
    return model
```

In [307]:
```
# Load dataset
X_train, y_train, X_test, y_test = load_dataset()# function to load ,reshape and normalize dataset
```

In [309]:
```
X_train.shape
```

Out[309]:
```
(60000, 28, 28, 1)
```

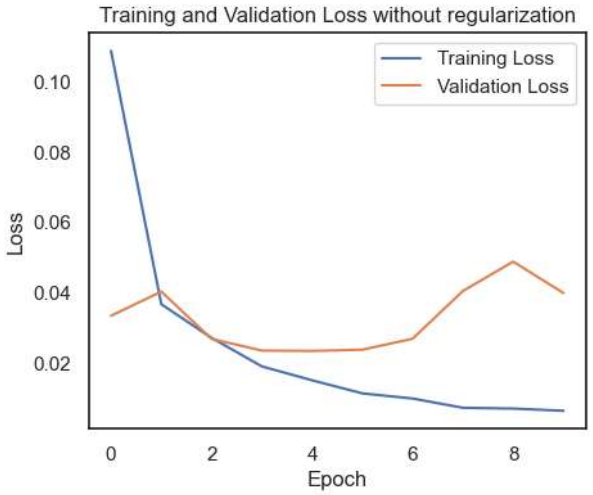
In [312]:
```
model = Non_reg_model()# model without regularisation
model.summary()# checking the summary of the model
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 28, 28, 32)        832

 max_pooling2d_4 (MaxPooling  (None, 14, 14, 32)       0
 2D)

 conv2d_5 (Conv2D)           (None, 14, 14, 64)        51264

 max_pooling2d_5 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 flatten_2 (Flatten)         (None, 3136)              0

 dense_4 (Dense)             (None, 128)               401536

 dense_5 (Dense)             (None, 10)                1290

=================================================================
Total params: 454,922
Trainable params: 454,922
Non-trainable params: 0
_________________________________________________________________
```

In [313]:
```
# Compiling the model (Learning rate=0.01,Optimizer=Adam,dropout=0,batch size=32,Epochs=10)
optimizer = Adam(learning_rate=0.001) # Adjusted learning rate to be more reasonable
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

# Training the model
history=model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=32)

# Evaluating the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Epoch 1/10
1875/1875 [==============================] - 181s 95ms/step - loss: 0.1089 - accuracy: 0.9661 - val_loss: 0.0334 - val_accurac
y: 0.9886
Epoch 2/10
1875/1875 [==============================] - 170s 91ms/step - loss: 0.0367 - accuracy: 0.9888 - val_loss: 0.0403 - val_accurac
y: 0.9875
Epoch 3/10
1875/1875 [==============================] - 173s 92ms/step - loss: 0.0271 - accuracy: 0.9914 - val_loss: 0.0268 - val_accurac
y: 0.9912
Epoch 4/10
1875/1875 [==============================] - 136s 72ms/step - loss: 0.0190 - accuracy: 0.9938 - val_loss: 0.0235 - val_accurac
y: 0.9918
Epoch 5/10
1875/1875 [==============================] - 139s 74ms/step - loss: 0.0150 - accuracy: 0.9949 - val_loss: 0.0234 - val_accurac
y: 0.9922
Epoch 6/10
1875/1875 [==============================] - 127s 68ms/step - loss: 0.0113 - accuracy: 0.9962 - val_loss: 0.0237 - val_accurac
y: 0.9924
Epoch 7/10
1875/1875 [==============================] - 127s 68ms/step - loss: 0.0098 - accuracy: 0.9969 - val_loss: 0.0269 - val_accurac
y: 0.9924
Epoch 8/10
1875/1875 [==============================] - 141s 75ms/step - loss: 0.0072 - accuracy: 0.9977 - val_loss: 0.0405 - val_accurac
y: 0.9902
Epoch 9/10
1875/1875 [==============================] - 126s 67ms/step - loss: 0.0070 - accuracy: 0.9978 - val_loss: 0.0488 - val_accurac
y: 0.9886
Epoch 10/10
1875/1875 [==============================] - 134s 72ms/step - loss: 0.0063 - accuracy: 0.9979 - val_loss: 0.0399 - val_accurac
y: 0.9914
Accuracy: 99.14%
```

In [317]:
```
# Extracting the loss values from the history
training_loss = history.history['loss']
validation_loss = history.history['val_loss']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(training_loss)), y=training_loss, label='Training Loss')
sns.lineplot(x=range(len(validation_loss)), y=validation_loss, label='Validation Loss')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss without regularization')
fig.savefig('Training and Validation Loss without regularization.png')
```



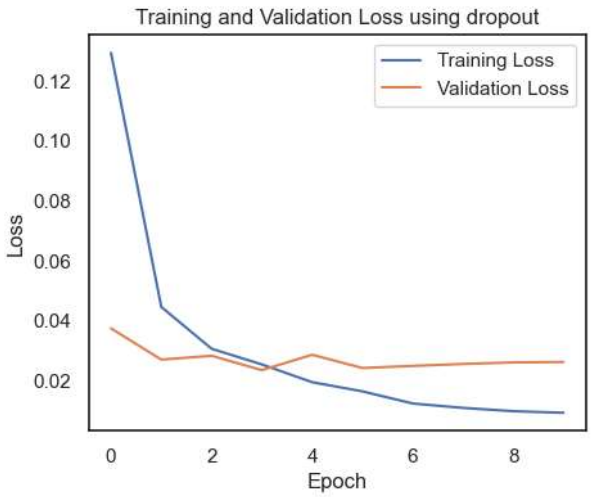
In [318]:
```
# Define the model
model = define_model()# model with 2 convolution blocks with dropout included
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 28, 28, 32)        832

 max_pooling2d_6 (MaxPooling  (None, 14, 14, 32)       0
 2D)

 conv2d_7 (Conv2D)           (None, 14, 14, 64)        51264

 max_pooling2d_7 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 flatten_3 (Flatten)         (None, 3136)              0

 dense_6 (Dense)             (None, 128)               401536

 dropout_2 (Dropout)         (None, 128)               0

 dense_7 (Dense)             (None, 10)                1290

=================================================================
Total params: 454,922
Trainable params: 454,922
Non-trainable params: 0
_________________________________________________________________
```

In [319]:
```
# Compile the model
optimizer = Adam(learning_rate=0.001) # Adjusted learning rate to be more reasonable
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

# Training the model
history=model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=32)

# Evaluating the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Epoch 1/10
1875/1875 [==============================] - 133s 71ms/step - loss: 0.1293 - accuracy: 0.9598 - val_loss: 0.0374 - val_accurac
y: 0.9881
Epoch 2/10
1875/1875 [==============================] - 121s 64ms/step - loss: 0.0445 - accuracy: 0.9862 - val_loss: 0.0269 - val_accurac
y: 0.9901
Epoch 3/10
1875/1875 [==============================] - 119s 63ms/step - loss: 0.0306 - accuracy: 0.9903 - val_loss: 0.0282 - val_accurac
y: 0.9907
Epoch 4/10
1875/1875 [==============================] - 127s 68ms/step - loss: 0.0253 - accuracy: 0.9919 - val_loss: 0.0234 - val_accurac
y: 0.9926
Epoch 5/10
1875/1875 [==============================] - 128s 68ms/step - loss: 0.0194 - accuracy: 0.9941 - val_loss: 0.0286 - val_accurac
y: 0.9909
Epoch 6/10
1875/1875 [==============================] - 135s 72ms/step - loss: 0.0163 - accuracy: 0.9950 - val_loss: 0.0241 - val_accurac
y: 0.9928
Epoch 7/10
1875/1875 [==============================] - 142s 76ms/step - loss: 0.0123 - accuracy: 0.9961 - val_loss: 0.0248 - val_accurac
y: 0.9932
Epoch 8/10
1875/1875 [==============================] - 129s 69ms/step - loss: 0.0108 - accuracy: 0.9966 - val_loss: 0.0255 - val_accurac
y: 0.9932
Epoch 9/10
1875/1875 [==============================] - 128s 68ms/step - loss: 0.0097 - accuracy: 0.9965 - val_loss: 0.0260 - val_accurac
y: 0.9932
Epoch 10/10
1875/1875 [==============================] - 129s 69ms/step - loss: 0.0092 - accuracy: 0.9972 - val_loss: 0.0261 - val_accurac
y: 0.9935
Accuracy: 99.35%
```

In [320]:
```
# Extracting the loss values from the history
training_loss = history.history['loss']
validation_loss = history.history['val_loss']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(training_loss)), y=training_loss, label='Training Loss')
sns.lineplot(x=range(len(validation_loss)), y=validation_loss, label='Validation Loss')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss using dropout');
fig.savefig('Training and Validation Loss with dropout.png')
```



from the plot above it can be observed that the validation loss increased as the number of epochs increased. This is therefore addressed by adopting regularisation techniques to reduce the complexity of the model. The following regularization techniques will be added

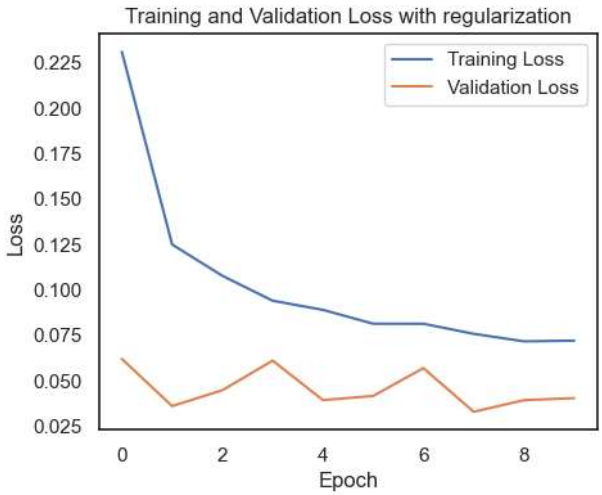
1. Data Augmentation-rotation,scaling,zooming
2. Early stopping
3. Hyper parameter tuning-learning rate,batch size and optimizer

Regularising Dataset

In [321]:
```
#Data Augmentation
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
    rotation_range=20,   # randomly rotate images up to 20 degrees
    width_shift_range=0.1,   # randomly shift images horizontally by up to 10%
    height_shift_range=0.1,   # randomly shift images vertically by up to 10%
    zoom_range=0.1,   # randomly zoom in or out on images by up to 10%
    horizontal_flip=True,   # randomly flip images horizontally
    fill_mode='nearest'   # fill any empty pixels with the nearest value
)
#Using Adam as the optimizer
history = model.fit(
    train_datagen.flow(X_train, y_train, batch_size=32),   # use the ImageDataGenerator to generate augmented images on the fly
    epochs=10,
    validation_data=(X_test, y_test)
)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Epoch 1/10
1875/1875 [==============================] - 145s 77ms/step - loss: 0.2310 - accuracy: 0.9299 - val_loss: 0.0620 - val_accurac
y: 0.9807
Epoch 2/10
1875/1875 [==============================] - 129s 69ms/step - loss: 0.1250 - accuracy: 0.9619 - val_loss: 0.0360 - val_accurac
y: 0.9877
Epoch 3/10
1875/1875 [==============================] - 127s 67ms/step - loss: 0.1077 - accuracy: 0.9669 - val_loss: 0.0448 - val_accurac
y: 0.9848
Epoch 4/10
1875/1875 [==============================] - 131s 70ms/step - loss: 0.0940 - accuracy: 0.9707 - val_loss: 0.0611 - val_accurac
y: 0.9812
Epoch 5/10
1875/1875 [==============================] - 127s 68ms/step - loss: 0.0890 - accuracy: 0.9724 - val_loss: 0.0393 - val_accurac
y: 0.9876
Epoch 6/10
1875/1875 [==============================] - 132s 71ms/step - loss: 0.0814 - accuracy: 0.9748 - val_loss: 0.0416 - val_accurac
y: 0.9861
Epoch 7/10
1875/1875 [==============================] - 149s 79ms/step - loss: 0.0813 - accuracy: 0.9745 - val_loss: 0.0569 - val_accurac
y: 0.9810
Epoch 8/10
1875/1875 [==============================] - 149s 79ms/step - loss: 0.0758 - accuracy: 0.9763 - val_loss: 0.0329 - val_accurac
y: 0.9892
Epoch 9/10
1875/1875 [==============================] - 146s 78ms/step - loss: 0.0717 - accuracy: 0.9778 - val_loss: 0.0393 - val_accurac
y: 0.9877
Epoch 10/10
1875/1875 [==============================] - 148s 79ms/step - loss: 0.0721 - accuracy: 0.9776 - val_loss: 0.0404 - val_accurac
y: 0.9875
Accuracy: 98.75%
```

In [324]:
```
# Extracting the loss values from the history
training_loss = history.history['loss']
validation_loss = history.history['val_loss']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(training_loss)), y=training_loss, label='Training Loss')
sns.lineplot(x=range(len(validation_loss)), y=validation_loss, label='Validation Loss')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss with regularization');
fig.savefig('Training and Validation Loss with regularization.png')
```



Although, accuracy dropped to 98.79% from 99.3% after data augmentation, the problem of overfitting was addressed, where the model performed well on

Using RMSprop as the optimizer and keeping all other parameters constant (Learning rate=0.001,Batch Size=32,Epochs=10, CNN blocks=2)

In [325]:
```
#RMSprop with learning rate(0.001),batch_size(32),Epochs(10),CNN blocks(2)
optimizer = RMSprop(learning_rate=0.001) # Adjusted learning rate to be more reasonable
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])


history = model.fit(
    train_datagen.flow(X_train, y_train, batch_size=32),   # use the ImageDataGenerator to generate augmented images on the fly
    epochs=10,
    validation_data=(X_test, y_test)
)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

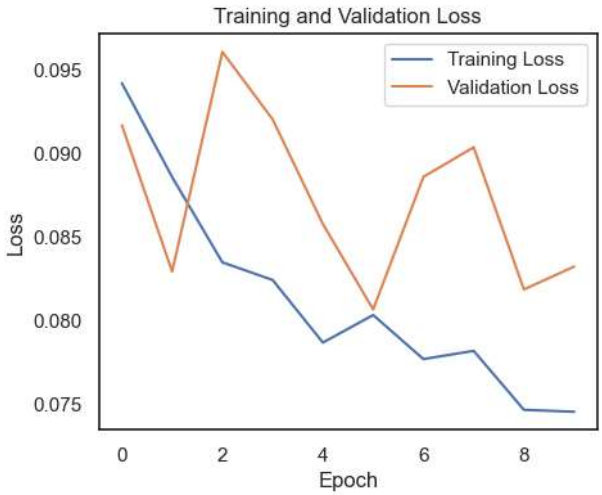
```
Epoch 1/10
1875/1875 [==============================] - 147s 78ms/step - loss: 0.0704 - accuracy: 0.9794 - val_loss: 0.0405 - val_accurac
y: 0.9889
Epoch 2/10
1875/1875 [==============================] - 135s 72ms/step - loss: 0.0686 - accuracy: 0.9797 - val_loss: 0.0429 - val_accurac
y: 0.9881
Epoch 3/10
1875/1875 [==============================] - 137s 73ms/step - loss: 0.0707 - accuracy: 0.9795 - val_loss: 0.0473 - val_accurac
y: 0.9879
Epoch 4/10
1875/1875 [==============================] - 136s 73ms/step - loss: 0.0724 - accuracy: 0.9789 - val_loss: 0.0563 - val_accurac
y: 0.9828
Epoch 5/10
1875/1875 [==============================] - 138s 74ms/step - loss: 0.0767 - accuracy: 0.9778 - val_loss: 0.0531 - val_accurac
y: 0.9840
Epoch 6/10
1875/1875 [==============================] - 142s 76ms/step - loss: 0.0822 - accuracy: 0.9773 - val_loss: 0.0597 - val_accurac
y: 0.9861
Epoch 7/10
1875/1875 [==============================] - 141s 75ms/step - loss: 0.0882 - accuracy: 0.9760 - val_loss: 0.0776 - val_accurac
y: 0.9872
Epoch 8/10
1875/1875 [==============================] - 124s 66ms/step - loss: 0.1017 - accuracy: 0.9730 - val_loss: 0.0777 - val_accurac
y: 0.9872
Epoch 9/10
1875/1875 [==============================] - 123s 66ms/step - loss: 0.1112 - accuracy: 0.9722 - val_loss: 0.1136 - val_accurac
y: 0.9846
Epoch 10/10
1875/1875 [==============================] - 137s 73ms/step - loss: 0.1223 - accuracy: 0.9699 - val_loss: 0.1129 - val_accurac
y: 0.9853
Accuracy: 98.53%
```

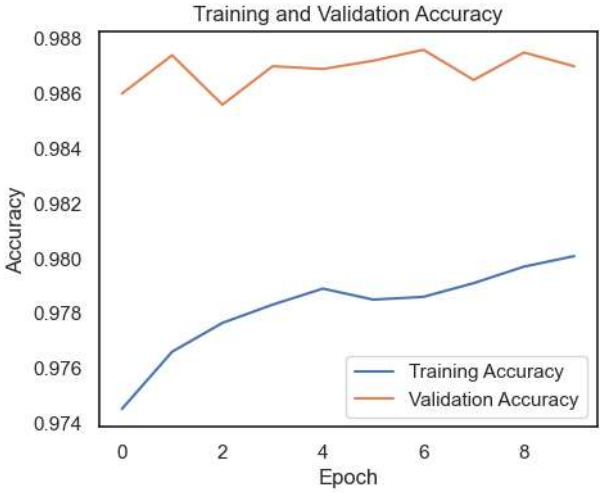
Using SGD as the optimizer and keeping all other parameters constant(Learning rate=0.001,Batch Size=32,Epochs=10, CNN blocks=2

In [330]:
```
# Compile the model
optimizer = SGD(learning_rate=0.001) # Adjusted learning rate to be more reasonable
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])


history = model.fit(
    train_datagen.flow(X_train, y_train, batch_size=32),   # use the ImageDataGenerator to generate augmented images on the fly
    epochs=10,
    validation_data=(X_test, y_test)
)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Epoch 1/10
1875/1875 [==============================] - 124s 66ms/step - loss: 0.0753 - accuracy: 0.9793 - val_loss: 0.0768 - val_accurac
y: 0.9860
Epoch 2/10
1875/1875 [==============================] - 116s 62ms/step - loss: 0.0766 - accuracy: 0.9793 - val_loss: 0.0819 - val_accurac
y: 0.9862
Epoch 3/10
1875/1875 [==============================] - 119s 63ms/step - loss: 0.0727 - accuracy: 0.9805 - val_loss: 0.0841 - val_accurac
y: 0.9868
Epoch 4/10
1875/1875 [==============================] - 140s 74ms/step - loss: 0.0718 - accuracy: 0.9803 - val_loss: 0.0968 - val_accurac
y: 0.9862
Epoch 5/10
1875/1875 [==============================] - 121s 65ms/step - loss: 0.0714 - accuracy: 0.9806 - val_loss: 0.0733 - val_accurac
y: 0.9869
Epoch 6/10
1875/1875 [==============================] - 119s 63ms/step - loss: 0.0716 - accuracy: 0.9806 - val_loss: 0.0780 - val_accurac
y: 0.9873
Epoch 7/10
1875/1875 [==============================] - 117s 62ms/step - loss: 0.0740 - accuracy: 0.9797 - val_loss: 0.0877 - val_accurac
y: 0.9875
Epoch 8/10
1875/1875 [==============================] - 119s 63ms/step - loss: 0.0714 - accuracy: 0.9804 - val_loss: 0.0840 - val_accurac
y: 0.9853
Epoch 9/10
1875/1875 [==============================] - 127s 68ms/step - loss: 0.0746 - accuracy: 0.9794 - val_loss: 0.0813 - val_accurac
y: 0.9874
Epoch 10/10
1875/1875 [==============================] - 127s 68ms/step - loss: 0.0694 - accuracy: 0.9813 - val_loss: 0.0867 - val_accurac
y: 0.9860
Accuracy: 98.60%
```

In [328]:
```
# Extracting the loss values from the history
training_loss = history.history['loss']
validation_loss = history.history['val_loss']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(training_loss)), y=training_loss, label='Training Loss')
sns.lineplot(x=range(len(validation_loss)), y=validation_loss, label='Validation Loss')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss');

# Extract the accuracy values from the history object
training_accuracy = history.history['accuracy']
validation_accuracy = history.history['val_accuracy']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(training_accuracy)), y=training_accuracy, label='Training Accuracy')
sns.lineplot(x=range(len(validation_accuracy)), y=validation_accuracy, label='Validation Accuracy')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy');
```





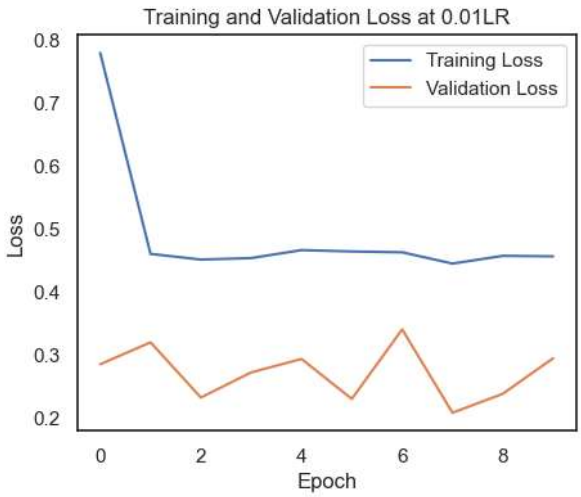
The accuracy improved from 98.79% to 98.93% after changing the optimizer to SGD

increasing the learning rate from 0.001 to 0.01 and keeping all other parameters constant(Optimizer=Adam,Batch Size=32,Epochs=10, CNN blocks=2)

In [351]:
```
# Compile the model
optimizer = Adam(learning_rate=0.01) # Adjusted learning rate to be more reasonable
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])


history = model.fit(
    train_datagen.flow(X_train, y_train, batch_size=32),   # use the ImageDataGenerator to generate augmented images on the fly
    epochs=10,
    validation_data=(X_test, y_test)
)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Epoch 1/10
1875/1875 [==============================] - 176s 93ms/step - loss: 0.7800 - accuracy: 0.7757 - val_loss: 0.2860 - val_accurac
y: 0.9124
Epoch 2/10
1875/1875 [==============================] - 159s 85ms/step - loss: 0.4611 - accuracy: 0.8635 - val_loss: 0.3209 - val_accurac
y: 0.9046
Epoch 3/10
1875/1875 [==============================] - 166s 89ms/step - loss: 0.4522 - accuracy: 0.8697 - val_loss: 0.2333 - val_accurac
y: 0.9274
Epoch 4/10
1875/1875 [==============================] - 170s 91ms/step - loss: 0.4545 - accuracy: 0.8689 - val_loss: 0.2732 - val_accurac
y: 0.9241
Epoch 5/10
1875/1875 [==============================] - 165s 88ms/step - loss: 0.4672 - accuracy: 0.8689 - val_loss: 0.2945 - val_accurac
y: 0.9156
Epoch 6/10
1875/1875 [==============================] - 164s 87ms/step - loss: 0.4650 - accuracy: 0.8719 - val_loss: 0.2311 - val_accurac
y: 0.9351
Epoch 7/10
1875/1875 [==============================] - 168s 90ms/step - loss: 0.4637 - accuracy: 0.8719 - val_loss: 0.3415 - val_accurac
y: 0.9116
Epoch 8/10
1875/1875 [==============================] - 165s 88ms/step - loss: 0.4458 - accuracy: 0.8770 - val_loss: 0.2091 - val_accurac
y: 0.9384
Epoch 9/10
1875/1875 [==============================] - 178s 95ms/step - loss: 0.4580 - accuracy: 0.8748 - val_loss: 0.2393 - val_accurac
y: 0.9346
Epoch 10/10
1875/1875 [==============================] - 171s 91ms/step - loss: 0.4573 - accuracy: 0.8755 - val_loss: 0.2956 - val_accurac
y: 0.9255
Accuracy: 92.55%
```

In [353]:
```
# Extracting the loss values from the history
training_loss = history.history['loss']
validation_loss = history.history['val_loss']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(training_loss)), y=training_loss, label='Training Loss')
sns.lineplot(x=range(len(validation_loss)), y=validation_loss, label='Validation Loss')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss at 0.01LR');
fig.savefig('Training and Validation Loss at 0.01LR.png')
```

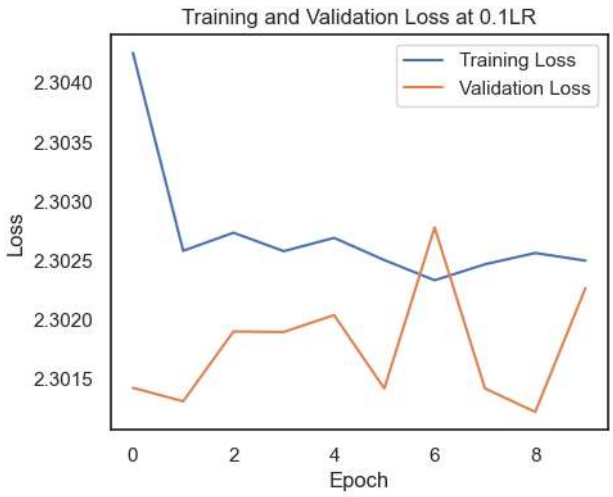


increasing the learning rate from 0.01 to 0.1 and keeping all other parameters constant(Optimizer=Adam,Batch Size=32,Epochs=10, CNN blocks=2)

In [354]:
```
optimizer = Adam(learning_rate=0.01) # Adjusted learning rate to be more reasonable
model=define_model()
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])


history = model.fit(
    train_datagen.flow(X_train, y_train, batch_size=32),   # use the ImageDataGenerator to generate augmented images on the fly
    epochs=10,
    validation_data=(X_test, y_test)
)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Epoch 1/10
1875/1875 [==============================] - 150s 79ms/step - loss: 2.3043 - accuracy: 0.1102 - val_loss: 2.3014 - val_accurac
y: 0.1135
Epoch 2/10
1875/1875 [==============================] - 146s 78ms/step - loss: 2.3026 - accuracy: 0.1088 - val_loss: 2.3013 - val_accurac
y: 0.1135
Epoch 3/10
1875/1875 [==============================] - 161s 86ms/step - loss: 2.3027 - accuracy: 0.1111 - val_loss: 2.3019 - val_accurac
y: 0.1028
Epoch 4/10
1875/1875 [==============================] - 141s 75ms/step - loss: 2.3026 - accuracy: 0.1099 - val_loss: 2.3019 - val_accurac
y: 0.1010
Epoch 5/10
1875/1875 [==============================] - 156s 83ms/step - loss: 2.3027 - accuracy: 0.1112 - val_loss: 2.3020 - val_accurac
y: 0.1028
Epoch 6/10
1875/1875 [==============================] - 156s 83ms/step - loss: 2.3025 - accuracy: 0.1092 - val_loss: 2.3014 - val_accurac
y: 0.1135
Epoch 7/10
1875/1875 [==============================] - 171s 91ms/step - loss: 2.3023 - accuracy: 0.1098 - val_loss: 2.3028 - val_accurac
y: 0.1135
Epoch 8/10
1875/1875 [==============================] - 186s 99ms/step - loss: 2.3025 - accuracy: 0.1105 - val_loss: 2.3014 - val_accurac
y: 0.1135
Epoch 9/10
1875/1875 [==============================] - 199s 106ms/step - loss: 2.3026 - accuracy: 0.1089 - val_loss: 2.3012 - val_accurac
y: 0.1135
Epoch 10/10
1875/1875 [==============================] - 192s 103ms/step - loss: 2.3025 - accuracy: 0.1081 - val_loss: 2.3023 - val_accurac
y: 0.1135
Accuracy: 11.35%
```

In [357]:
```
# Extracting the loss values from the history
training_loss = history.history['loss']
validation_loss = history.history['val_loss']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(training_loss)), y=training_loss, label='Training Loss')
sns.lineplot(x=range(len(validation_loss)), y=validation_loss, label='Validation Loss')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss at 0.1LR');
fig.savefig('Training and Validation Loss at 0.1LR.png')
```



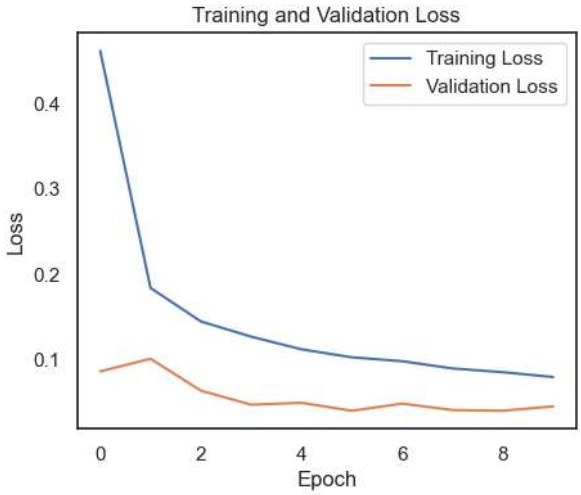
There was a drop in the accuracy from 98.75% when learning rate was 0.001 to 92.55% when learning rate was changed to 0.01. There was a further decline in accuracy rate to 11.35% when learning rate was 0.1 hence an increase in learning rate did not improve the performance of the model.

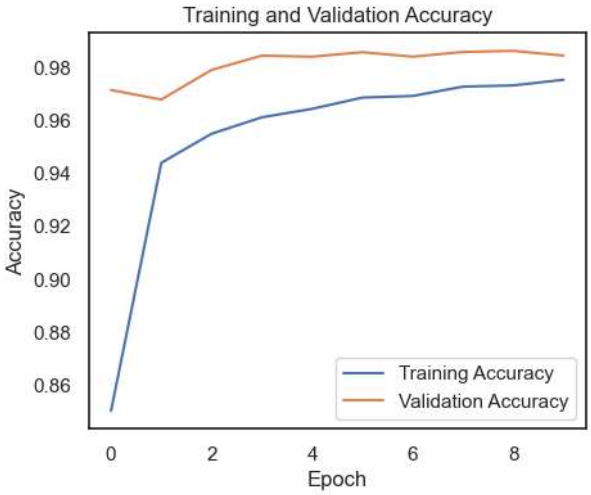
Increasing the batch size to 64, reverting to the initial learning rate(0.001) and keeping all other parameters constant(Optimizer=Adam,learning rate=0.001,Epochs=10, CNN blocks=2))

In [340]:
```
# Compile the model
model=define_model()
optimizer = Adam(learning_rate=0.001) # Adjusted learning rate to be more reasonable
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])


history = model.fit(
    train_datagen.flow(X_train, y_train, batch_size=64),   # use the ImageDataGenerator to generate augmented images on the fly
    epochs=10,
    validation_data=(X_test, y_test)
)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Epoch 1/10
938/938 [==============================] - 125s 132ms/step - loss: 0.4615 - accuracy: 0.8502 - val_loss: 0.0859 - val_accuracy:
0.9716
Epoch 2/10
938/938 [==============================] - 124s 132ms/step - loss: 0.1836 - accuracy: 0.9441 - val_loss: 0.1007 - val_accuracy:
0.9680
Epoch 3/10
938/938 [==============================] - 118s 125ms/step - loss: 0.1444 - accuracy: 0.9550 - val_loss: 0.0633 - val_accuracy:
0.9792
Epoch 4/10
938/938 [==============================] - 121s 129ms/step - loss: 0.1265 - accuracy: 0.9613 - val_loss: 0.0468 - val_accuracy:
0.9846
Epoch 5/10
938/938 [==============================] - 116s 124ms/step - loss: 0.1117 - accuracy: 0.9645 - val_loss: 0.0490 - val_accuracy:
0.9842
Epoch 6/10
938/938 [==============================] - 118s 125ms/step - loss: 0.1024 - accuracy: 0.9687 - val_loss: 0.0397 - val_accuracy:
0.9859
Epoch 7/10
938/938 [==============================] - 114s 121ms/step - loss: 0.0978 - accuracy: 0.9693 - val_loss: 0.0480 - val_accuracy:
0.9842
Epoch 8/10
938/938 [==============================] - 119s 127ms/step - loss: 0.0893 - accuracy: 0.9729 - val_loss: 0.0406 - val_accuracy:
0.9860
Epoch 9/10
938/938 [==============================] - 119s 126ms/step - loss: 0.0849 - accuracy: 0.9733 - val_loss: 0.0397 - val_accuracy:
0.9864
Epoch 10/10
938/938 [==============================] - 117s 125ms/step - loss: 0.0793 - accuracy: 0.9755 - val_loss: 0.0449 - val_accuracy:
0.9846
Accuracy: 98.46%
```

In [341]:
```
# Extract the loss values from the history object
train_loss = history.history['loss']
val_loss = history.history['val_loss']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(train_loss)), y=train_loss, label='Training Loss')
sns.lineplot(x=range(len(val_loss)), y=val_loss, label='Validation Loss')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss');


# Extract the accuracy values from the history object
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(train_accuracy)), y=train_accuracy, label='Training Accuracy')
sns.lineplot(x=range(len(val_accuracy)), y=val_accuracy, label='Validation Accuracy')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy');
```





Using Adam optimizer, a learning rate of 0,001, batch size of 32, epoch 10 and increasing the number of convolution blocks to 3

In [342]:
```
# Building cnn model with 3 convolutional blocks
def define_model3():#Model with 3 CNN blocks
    model = Sequential()
    model.add(Conv2D(filters=32, kernel_size=(5, 5), input_shape=(28, 28, 1), activation='relu', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(filters=64, kernel_size=(5, 5), activation='relu', padding='same'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(filters=64, kernel_size=(5, 5), activation='relu', padding='same'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(10, activation='softmax')) #the possible different classes
    return model
```

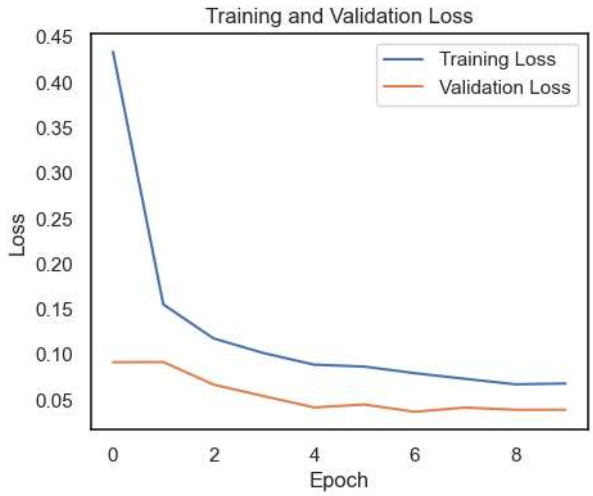
In [345]:
```
model=define_model3()# Model with 3 convolutional blocks
model.summary()
```

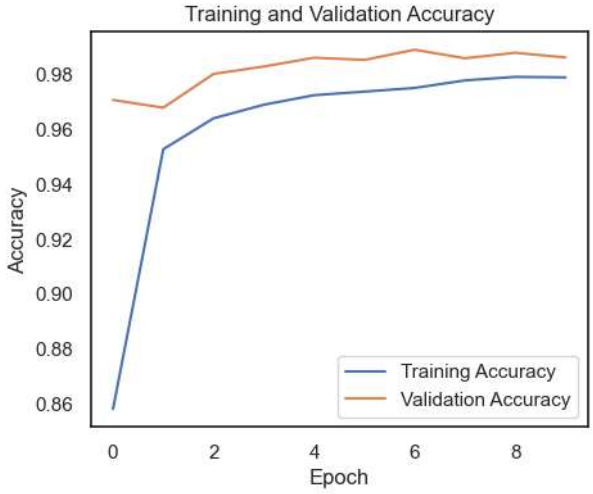
```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_14 (Conv2D)          (None, 28, 28, 32)        832

 max_pooling2d_14 (MaxPoolin  (None, 14, 14, 32)       0
 g2D)

 conv2d_15 (Conv2D)          (None, 14, 14, 64)        51264

 max_pooling2d_15 (MaxPoolin  (None, 7, 7, 64)         0
 g2D)

 conv2d_16 (Conv2D)          (None, 7, 7, 64)          102464

 max_pooling2d_16 (MaxPoolin  (None, 3, 3, 64)         0
 g2D)

 flatten_7 (Flatten)         (None, 576)               0

 dense_14 (Dense)            (None, 128)               73856

 dropout_6 (Dropout)         (None, 128)               0

 dense_15 (Dense)            (None, 10)                1290

=================================================================
Total params: 229,706
Trainable params: 229,706
Non-trainable params: 0
_________________________________________________________________
```

In [346]:
```
# Compile the model
optimizer = Adam(learning_rate=0.001) # Adjusted learning rate to be more reasonable
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])


history = model.fit(
    train_datagen.flow(X_train, y_train, batch_size=64),   # use the ImageDataGenerator to generate augmented images on the fly
    epochs=10,
    validation_data=(X_test, y_test)
)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Epoch 1/10
938/938 [==============================] - 159s 167ms/step - loss: 0.4337 - accuracy: 0.8582 - val_loss: 0.0916 - val_accuracy:
0.9708
Epoch 2/10
938/938 [==============================] - 162s 172ms/step - loss: 0.1553 - accuracy: 0.9528 - val_loss: 0.0918 - val_accuracy:
0.9680
Epoch 3/10
938/938 [==============================] - 163s 174ms/step - loss: 0.1178 - accuracy: 0.9641 - val_loss: 0.0669 - val_accuracy:
0.9803
Epoch 4/10
938/938 [==============================] - 160s 171ms/step - loss: 0.1016 - accuracy: 0.9691 - val_loss: 0.0542 - val_accuracy:
0.9830
Epoch 5/10
938/938 [==============================] - 149s 159ms/step - loss: 0.0889 - accuracy: 0.9725 - val_loss: 0.0417 - val_accuracy:
0.9862
Epoch 6/10
938/938 [==============================] - 145s 154ms/step - loss: 0.0868 - accuracy: 0.9739 - val_loss: 0.0449 - val_accuracy:
0.9854
Epoch 7/10
938/938 [==============================] - 146s 156ms/step - loss: 0.0795 - accuracy: 0.9752 - val_loss: 0.0370 - val_accuracy:
0.9891
Epoch 8/10
938/938 [==============================] - 145s 155ms/step - loss: 0.0733 - accuracy: 0.9779 - val_loss: 0.0416 - val_accuracy:
0.9860
Epoch 9/10
938/938 [==============================] - 142s 152ms/step - loss: 0.0673 - accuracy: 0.9792 - val_loss: 0.0391 - val_accuracy:
0.9880
Epoch 10/10
938/938 [==============================] - 151s 161ms/step - loss: 0.0681 - accuracy: 0.9790 - val_loss: 0.0392 - val_accuracy:
0.9863
Accuracy: 98.63%
```

In [347]:
```
# Extracting the loss values from the history
train_loss = history.history['loss']
val_loss = history.history['val_loss']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(train_loss)), y=train_loss, label='Training Loss')
sns.lineplot(x=range(len(val_loss)), y=val_loss, label='Validation Loss')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss');

# Extract the accuracy values from the history object
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(train_accuracy)), y=train_accuracy, label='Training Accuracy')
sns.lineplot(x=range(len(val_accuracy)), y=val_accuracy, label='Validation Accuracy')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy');
```





increasing the number of convolution blocks didn't improve the accuracy; rather,there was a slight reduction in accuracy by 0.01%(99.03% to 99.02%). hence, an increase in the number of convolution blocks does not increase the performance of the model.

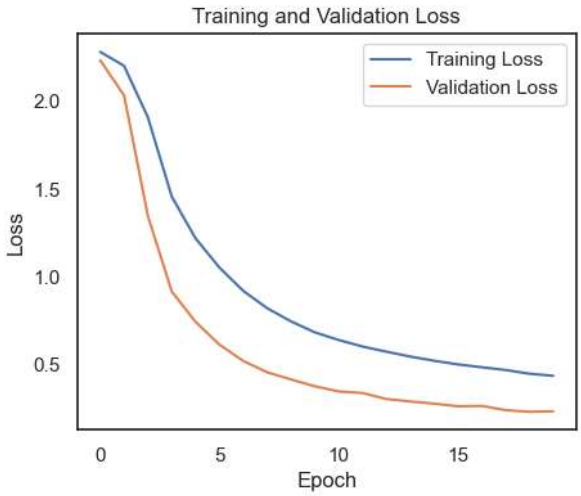
Using CNN block of 3,learning rate of 0.001,batch size of 64,epoch of 20 and a Adam optimizer

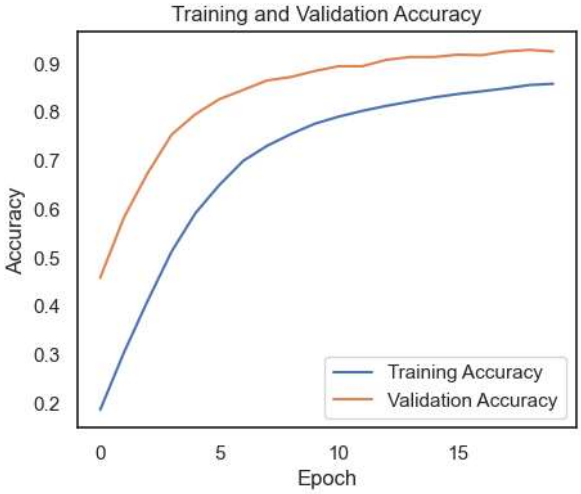
In [348]:

```
# Compile the model
model=define_model3()# model with 3 CNN blocks
optimizer = SGD(learning_rate=0.001) # Adjusted learning rate to be more reasonable
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])


history = model.fit(
    train_datagen.flow(X_train, y_train, batch_size=64),   # use the ImageDataGenerator to generate augmented images on the fly
    epochs=20,
    validation_data=(X_test, y_test)
)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Epoch 1/20
938/938 [==============================] - 173s 184ms/step - loss: 2.2814 - accuracy: 0.1861 - val_loss: 2.2331 - val_accuracy:
0.4583
Epoch 2/20
938/938 [==============================] - 151s 161ms/step - loss: 2.2020 - accuracy: 0.3046 - val_loss: 2.0340 - val_accuracy:
0.5832
Epoch 3/20
938/938 [==============================] - 150s 160ms/step - loss: 1.9088 - accuracy: 0.4119 - val_loss: 1.3475 - val_accuracy:
0.6751
Epoch 4/20
938/938 [==============================] - 142s 152ms/step - loss: 1.4576 - accuracy: 0.5130 - val_loss: 0.9168 - val_accuracy:
0.7538
Epoch 5/20
938/938 [==============================] - 150s 160ms/step - loss: 1.2190 - accuracy: 0.5920 - val_loss: 0.7434 - val_accuracy:
0.7959
Epoch 6/20
938/938 [==============================] - 144s 154ms/step - loss: 1.0530 - accuracy: 0.6498 - val_loss: 0.6141 - val_accuracy:
0.8267
Epoch 7/20
938/938 [==============================] - 145s 155ms/step - loss: 0.9200 - accuracy: 0.6995 - val_loss: 0.5210 - val_accuracy:
0.8458
Epoch 8/20
938/938 [==============================] - 148s 158ms/step - loss: 0.8217 - accuracy: 0.7305 - val_loss: 0.4565 - val_accuracy:
0.8654
Epoch 9/20
938/938 [==============================] - 160s 170ms/step - loss: 0.7466 - accuracy: 0.7547 - val_loss: 0.4156 - val_accuracy:
0.8725
Epoch 10/20
938/938 [==============================] - 161s 172ms/step - loss: 0.6841 - accuracy: 0.7760 - val_loss: 0.3766 - val_accuracy:
0.8848
Epoch 11/20
938/938 [==============================] - 154s 164ms/step - loss: 0.6403 - accuracy: 0.7907 - val_loss: 0.3472 - val_accuracy:
0.8947
Epoch 12/20
938/938 [==============================] - 148s 158ms/step - loss: 0.6031 - accuracy: 0.8026 - val_loss: 0.3382 - val_accuracy:
0.8948
Epoch 13/20
938/938 [==============================] - 139s 148ms/step - loss: 0.5737 - accuracy: 0.8129 - val_loss: 0.3040 - val_accuracy:
0.9078
Epoch 14/20
938/938 [==============================] - 154s 164ms/step - loss: 0.5461 - accuracy: 0.8217 - val_loss: 0.2902 - val_accuracy:
0.9137
Epoch 15/20
938/938 [==============================] - 155s 165ms/step - loss: 0.5220 - accuracy: 0.8304 - val_loss: 0.2778 - val_accuracy:
0.9137
Epoch 16/20
938/938 [==============================] - 164s 175ms/step - loss: 0.5012 - accuracy: 0.8374 - val_loss: 0.2627 - val_accuracy:
0.9185
Epoch 17/20
938/938 [==============================] - 156s 167ms/step - loss: 0.4851 - accuracy: 0.8430 - val_loss: 0.2648 - val_accuracy:
0.9174
Epoch 18/20
938/938 [==============================] - 153s 163ms/step - loss: 0.4694 - accuracy: 0.8490 - val_loss: 0.2409 - val_accuracy:
0.9251
Epoch 19/20
938/938 [==============================] - 155s 166ms/step - loss: 0.4483 - accuracy: 0.8559 - val_loss: 0.2313 - val_accuracy:
0.9284
Epoch 20/20
938/938 [==============================] - 144s 153ms/step - loss: 0.4363 - accuracy: 0.8584 - val_loss: 0.2343 - val_accuracy:
0.9251
Accuracy: 92.51%
```

In [349]:
```
# Extract the loss values from the history object
train_loss = history.history['loss']
val_loss = history.history['val_loss']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(train_loss)), y=train_loss, label='Training Loss')
sns.lineplot(x=range(len(val_loss)), y=val_loss, label='Validation Loss')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss');

# Extract the accuracy values from the history object
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(train_accuracy)), y=train_accuracy, label='Training Accuracy')
sns.lineplot(x=range(len(val_accuracy)), y=val_accuracy, label='Validation Accuracy')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy');
```



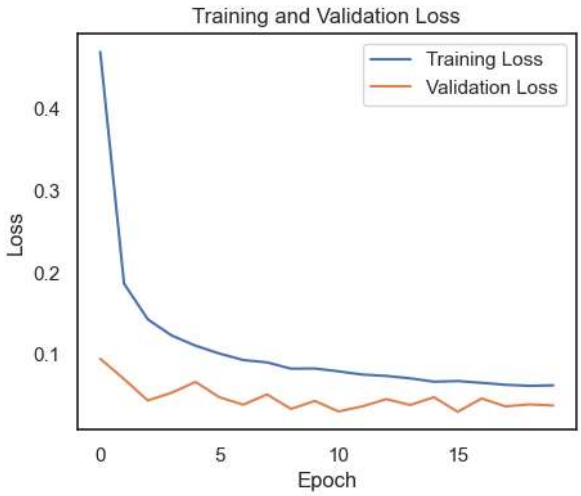


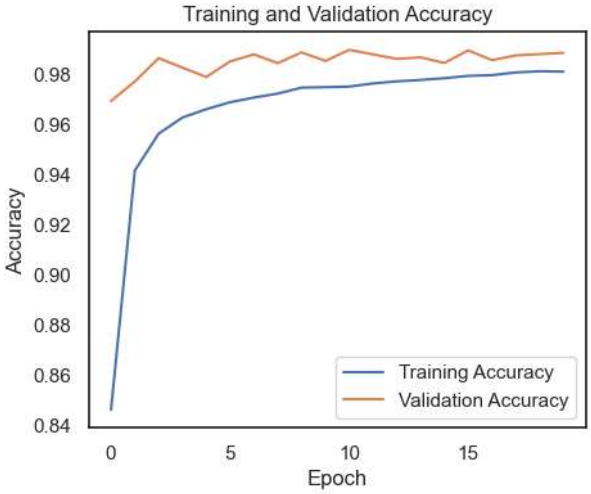
Using CNN block of 3,learning rate of 0.001,batch size of 64,epoch of 20 and an Adam optimizer

In [367]:
```
# Compile the model
optimizer = Adam(learning_rate=0.001) # Adjusted learning rate to be more reasonable
model=define_model()#CNN block of 2
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])


history = model.fit(
    train_datagen.flow(X_train, y_train, batch_size=64),   # use the ImageDataGenerator to generate augmented images on the fly
    epochs=20,
    validation_data=(X_test, y_test)
)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Epoch 1/20
938/938 [==============================] - 117s 123ms/step - loss: 0.4694 - accuracy: 0.8461 - val_loss: 0.0949 - val_accuracy:
0.9692
Epoch 2/20
938/938 [==============================] - 117s 125ms/step - loss: 0.1865 - accuracy: 0.9415 - val_loss: 0.0700 - val_accuracy:
0.9770
Epoch 3/20
938/938 [==============================] - 120s 128ms/step - loss: 0.1428 - accuracy: 0.9562 - val_loss: 0.0439 - val_accuracy:
0.9863
Epoch 4/20
938/938 [==============================] - 120s 127ms/step - loss: 0.1232 - accuracy: 0.9626 - val_loss: 0.0534 - val_accuracy:
0.9825
Epoch 5/20
938/938 [==============================] - 116s 123ms/step - loss: 0.1108 - accuracy: 0.9660 - val_loss: 0.0665 - val_accuracy:
0.9788
Epoch 6/20
938/938 [==============================] - 121s 129ms/step - loss: 0.1012 - accuracy: 0.9687 - val_loss: 0.0478 - val_accuracy:
0.9850
Epoch 7/20
938/938 [==============================] - 120s 128ms/step - loss: 0.0933 - accuracy: 0.9706 - val_loss: 0.0388 - val_accuracy:
0.9878
Epoch 8/20
938/938 [==============================] - 119s 127ms/step - loss: 0.0904 - accuracy: 0.9722 - val_loss: 0.0513 - val_accuracy:
0.9843
Epoch 9/20
938/938 [==============================] - 115s 123ms/step - loss: 0.0827 - accuracy: 0.9746 - val_loss: 0.0336 - val_accuracy:
0.9886
Epoch 10/20
938/938 [==============================] - 124s 132ms/step - loss: 0.0830 - accuracy: 0.9747 - val_loss: 0.0433 - val_accuracy:
0.9852
Epoch 11/20
938/938 [==============================] - 124s 132ms/step - loss: 0.0795 - accuracy: 0.9750 - val_loss: 0.0305 - val_accuracy:
0.9896
Epoch 12/20
938/938 [==============================] - 121s 129ms/step - loss: 0.0755 - accuracy: 0.9762 - val_loss: 0.0367 - val_accuracy:
0.9878
Epoch 13/20
938/938 [==============================] - 143s 152ms/step - loss: 0.0739 - accuracy: 0.9771 - val_loss: 0.0455 - val_accuracy:
0.9860
Epoch 14/20
938/938 [==============================] - 137s 146ms/step - loss: 0.0709 - accuracy: 0.9776 - val_loss: 0.0384 - val_accuracy:
0.9866
Epoch 15/20
938/938 [==============================] - 141s 151ms/step - loss: 0.0668 - accuracy: 0.9783 - val_loss: 0.0479 - val_accuracy:
0.9844
Epoch 16/20
938/938 [==============================] - 132s 141ms/step - loss: 0.0676 - accuracy: 0.9792 - val_loss: 0.0301 - val_accuracy:
0.9894
Epoch 17/20
938/938 [==============================] - 126s 134ms/step - loss: 0.0654 - accuracy: 0.9795 - val_loss: 0.0464 - val_accuracy:
0.9855
Epoch 18/20
938/938 [==============================] - 131s 139ms/step - loss: 0.0631 - accuracy: 0.9806 - val_loss: 0.0367 - val_accuracy:
0.9874
Epoch 19/20
938/938 [==============================] - 126s 134ms/step - loss: 0.0618 - accuracy: 0.9811 - val_loss: 0.0391 - val_accuracy:
0.9879
Epoch 20/20
938/938 [==============================] - 116s 123ms/step - loss: 0.0624 - accuracy: 0.9809 - val_loss: 0.0378 - val_accuracy:
0.9884
Accuracy: 98.84%
```

In [368]:
```
# Extract the loss values from the history object
train_loss = history.history['loss']
val_loss = history.history['val_loss']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(train_loss)), y=train_loss, label='Training Loss')
sns.lineplot(x=range(len(val_loss)), y=val_loss, label='Validation Loss')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss');

# Extract the accuracy values from the history object
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']
fig, ax = plt.subplots(figsize=(5, 4))

# Create a line plot using Seaborn
sns.lineplot(x=range(len(train_accuracy)), y=train_accuracy, label='Training Accuracy')
sns.lineplot(x=range(len(val_accuracy)), y=val_accuracy, label='Validation Accuracy')

# Set plot labels and title
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy');
```

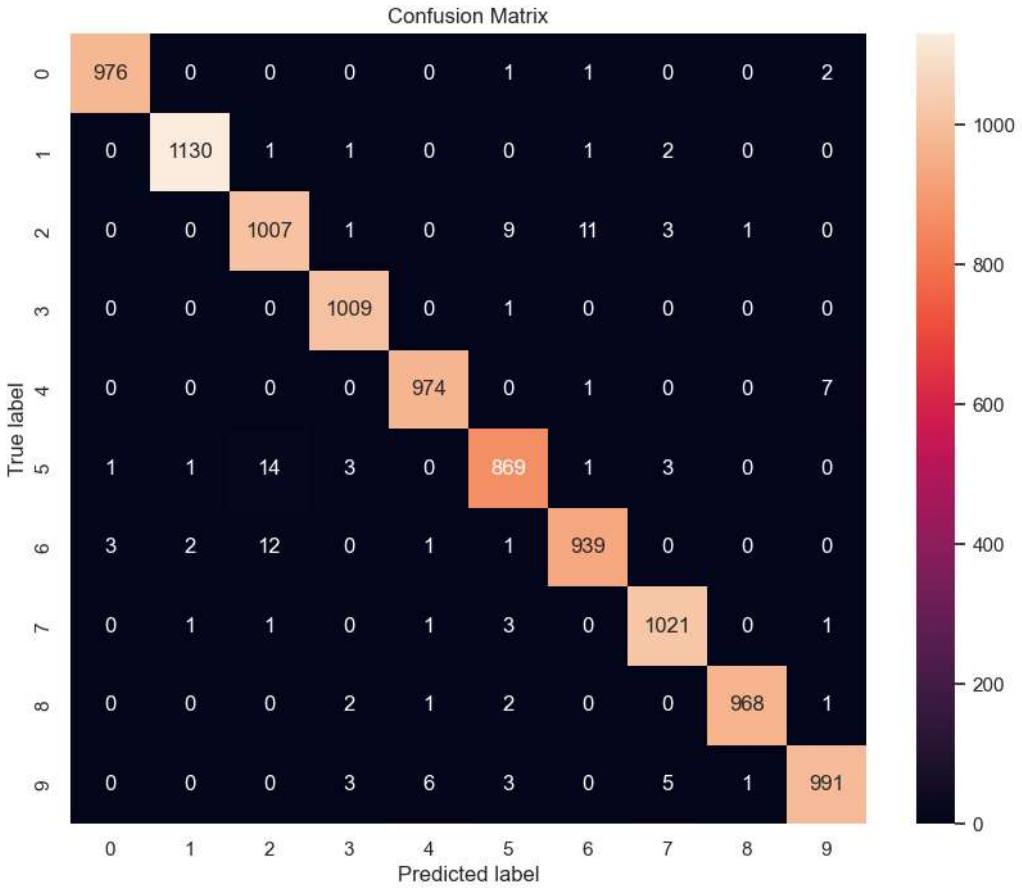




In [369]:
```
#Printing the classification report of the best model
y_pred = model.predict(X_test)
y_pred = np.argmax(y_pred, axis=1)

# Convert the one-hot encoded test labels to integer format
y_true = np.argmax(y_test, axis=1)

# Print the classification report
from sklearn.metrics import classification_report
report = classification_report(y_true, y_pred)
print(report)
```

```
313/313 [==============================] - 6s 17ms/step
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       980
           1       1.00      1.00      1.00      1135
           2       0.97      0.98      0.97      1032
           3       0.99      1.00      0.99      1010
           4       0.99      0.99      0.99       982
           5       0.98      0.97      0.98       892
           6       0.98      0.98      0.98       958
           7       0.99      0.99      0.99      1028
           8       1.00      0.99      1.00       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 [376]:
```
# Plotting the confusion matrix
cm = confusion_matrix(y_true, y_pred)
# Plot the confusion matrix using Seaborn
fig, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(cm, annot=True,fmt='g')
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.title('Confusion Matrix')
plt.show()
fig.savefig('Confusion Matrix plot.png')
```



In [ ]:
```
```