In [2]:
```
import pandas as pd
import re
import numpy as np
import nltk
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import LabelEncoder, FunctionTransformer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from scipy.stats import randint, uniform
from xgboost import XGBClassifier
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.probability import FreqDist
from nltk.collocations import BigramAssocMeasures, BigramCollocationFinder
from textblob import TextBlob, Blobber
from textblob.sentiments import NaiveBayesAnalyzer
from gensim.utils import simple_preprocess
from gensim.parsing.preprocessing import STOPWORDS
from gensim.corpora import Dictionary
from gensim.models import LdaModel
from gensim.models.phrases import Phrases
from collections import Counter
import random

import seaborn as sns
import matplotlib.pyplot as plt
import warnings

warnings.filterwarnings('ignore')
```

These are the libraries required for our code, including data processing, machine learning, NLP, and visualization libraries.

In [3]:
```python
#Import the Data
df = pd.read_csv('D:/Git/phase_4/Hades_reviews.csv')

#Check the Data
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 457440 entries, 0 to 457439
Data columns (total 27 columns):
 #   Column                          Non-Null Count   Dtype
---  ------                          --------------   -----
 0   Unnamed: 0                      457440 non-null  int64
 1   query_summary                   0 non-null       float64
 2   cursors                         0 non-null       float64
 3   recommendationid                228720 non-null  float64
 4   language                        228720 non-null  object
 5   review                          228017 non-null  object
 6   timestamp_created               228720 non-null  float64
 7   timestamp_updated               228720 non-null  float64
 8   voted_up                        228720 non-null  object
 9   votes_up                        228720 non-null  float64
 10  votes_funny                     228720 non-null  float64
 11  weighted_vote_score             228720 non-null  float64
 12  comment_count                   228720 non-null  float64
 13  steam_purchase                  228720 non-null  object
```

Data Cleaning

In [4]:
```python
#Drop Nulls
df = df.dropna(subset=['review'])

#Keep only English reviews
df = df[df['language'] == 'english']

# Drop Unnecessary Columns
df = df.drop(df.columns[[0, 1, 2, 3, 4, 6, 7, 16, 17, 18]], axis=1)

# Create a mask where each review has more than 5 words and at least one alphabetic character
mask = df['review'].apply(lambda x: len(re.findall(r'\b\w+\b', str(x))) > 5 and bool(re.search('

# Apply the mask to the DataFrame to filter out review
df = df[mask]
```

These lines drop the rows with missing values in the 'review' column, filter the DataFrame to keep only English reviews, and drop unnecessary columns from the DataFrame.

In [5]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 76744 entries, 228720 to 457437
Data columns (total 17 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   review                          76744 non-null  object
 1   voted_up                        76744 non-null  object
 2   votes_up                        76744 non-null  float64
 3   votes_funny                     76744 non-null  float64
 4   weighted_vote_score             76744 non-null  float64
 5   comment_count                   76744 non-null  float64
 6   steam_purchase                  76744 non-null  object
 7   received_for_free               76744 non-null  object
 8   written_during_early_access     76744 non-null  object
 9   author.num_games_owned          76744 non-null  float64
 10  author.num_reviews              76744 non-null  float64
 11  author.playtime_forever         76744 non-null  float64
 12  author.playtime_last_two_weeks  76744 non-null  float64
 13  author.playtime_at_review       76744 non-null  float64
 14  author.last_played              76744 non-null  float64
 15  timestamp_dev_responded         12 non-null     float64
 16  developer_response              12 non-null     object
dtypes: float64(11), object(6)
memory usage: 10.5+ MB
```

In [6]:
```
df.head()
```

Out[6]:

	review	voted_up	votes_up	votes_funny	weighted_vote_score	comment_count	steam_purchase	re
228720	Beautiful art and music, fun gameplay and grea...	True	0.0	0.0	0.0	0.0	True	
228721	Hades has a lot going for it the soundtrack, v...	True	0.0	0.0	0.0	0.0	True	
228723	perfect loop, beautiful art, fun weapons	True	0.0	0.0	0.0	0.0	True	
228724	Combat : 10/10\nReplayabilty : 10/10\nStory + ...	True	0.0	0.0	0.0	0.0	True	
228726	fun but u die alot LOL	True	0.0	0.0	0.0	0.0	False	

Step 1: Exploratory Data Analysis

These lines perform some EDA on the DataFrame, such as counting the number of positive and negative reviews, describing the playtime of the authors, calculating the length of each review, and providing summary statistics for the review length.

In [7]:
```
df['voted_up'].value_counts()
```

Out[7]:
```
True     75508
False     1236
Name: voted_up, dtype: int64
```

With all the "positive" reviews listed here ('voted_up') our data set will be extremely imbalanced if we focus on targeting whether a review was positive or note. So let's consider some other features.

In [8]:
```
df['author.playtime_forever'].describe()
```

Out[8]:
```
count     76744.000000
mean       5169.432190
std        6119.080535
min           5.000000
25%        1859.000000
50%        3914.000000
75%        6598.000000
max      272341.000000
Name: author.playtime_forever, dtype: float64
```

In [9]:
```
# Calculate the length of each review (in words)
df['review_length'] = df['review'].apply(lambda x: len(x.split()))

# Calculate the average length of reviews
average_length = df['review_length'].mean()

df['review_length'].describe()
```

Out[9]:
```
count    76744.000000
mean        48.357474
std         85.016701
min          1.000000
25%         11.000000
50%         22.000000
75%         50.000000
max       1600.000000
Name: review_length, dtype: float64
```

It looks like there is a nice spread in terms of play time and the length of reviews. Those might help us create a model with something to learn from.

Text preprocessing:

In [10]:
```
# Get list of stopwords
stop_words = set(stopwords.words('english'))

# Initialize a lemmatizer
lemmatizer = WordNetLemmatizer()

#Setup lemmatizer
def lemmatize_text(text):
    words = word_tokenize(text)
    filtered_words = [lemmatizer.lemmatize(w) for w in words if w.lower() not in stop_words]
    return ' '.join(filtered_words)

# Lemmatize the reviews
df['review'] = df['review'].apply(lemmatize_text)
```

These lines define a function lemmatize_text to lemmatize the review texts by removing stopwords and performing lemmatization. The function is then applied to the 'review' column using df['review'].apply()

In [11]:
```
# Encode review length into categories based on specific ranges or thresholds
df['review_length_category'] = pd.cut(df['review_length'], bins=[0, 8, 18, 44, np.inf], labels=[(

#Check value counts
df['review_length_category'].value_counts()
```

Out[11]:
```
3    21715
1    21613
2    21249
0    12167
Name: review_length_category, dtype: int64
```

This looks like a much more even spread! This should work as a variable.

In [13]:
```
# Calculate average playtime
average_playtime = df['author.playtime_forever'].mean()

# Create new binary column
df['above_average_playtime'] = np.where(df['author.playtime_forever'] > average_playtime, 1, 0)

#Check value counts
df['above_average_playtime'].value_counts()
```

Out[13]:
```
0    48192
1    28552
Name: above_average_playtime, dtype: int64
```

These lines encode the review length into categories based on specific ranges or thresholds and create a binary column indicating whether the playtime is above average or not.

Preprocessing pipeline and model training

In [15]:
```
# Define preprocessing for text column
text_features = 'review'
text_transformer = Pipeline(steps=[
    ('tfidf', TfidfVectorizer(max_features=1000))
])

# Define preprocessing for numeric column
numeric_features = ['above_average_playtime']
numeric_transformer = Pipeline(steps=[
    ('identity', FunctionTransformer(validate=False))  # Identity function - does nothing
])

# Combine preprocessing steps
preprocessor = ColumnTransformer(
    transformers=[
        ('text', text_transformer, text_features),
        ('num', numeric_transformer, numeric_features)
    ])

# Append classifier to preprocessing pipeline
clf = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier())
])
```

These lines define a preprocessing pipeline using ColumnTransformer to apply TF-IDF vectorization to the 'review' column and keep the 'above_average_playtime' column as numeric features. It then builds a pipeline with a RandomForestClassifier as the classifier. The data is split into training and testing sets using train_test_split, and the model is trained and evaluated using the classification report.

Now let's do a quick test of our data to see if we were right about the positive review prediction leading to overfitting due to an imbalanced data set. We will start with a simple *logistic regression model*:

In [16]:
```
# Define features and target for Logistic Regression model
X = df['review'].tolist()

# Get the labels (positive or negative)
y = df['voted_up'].map({True: 1, False: 0})

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [17]:
```
# Set up the pipeline for the Logistic Regression model
logreg_pipeline = Pipeline(steps=[
    ('tfidf', TfidfVectorizer(max_features=1000)),
    ('classifier', LogisticRegression())
])

# Fit the model and make predictions
logreg_pipeline.fit(X_train, y_train)
y_pred = logreg_pipeline.predict(X_test)

# Print classification report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.87      0.11      0.19       246
           1       0.99      1.00      0.99     15103

    accuracy                           0.99     15349
   macro avg       0.93      0.55      0.59     15349
weighted avg       0.98      0.99      0.98     15349
```

As predicted, our model is overfitting. So let's change tactics for our more complex models. Rather than trying to predict the positivity of a review based on its content, let's see if we can predict the length of a review by whether or not a player plays an above or below average amount. Because our data set is so large, we will only use a subset of the total data.

In [18]:
```
# Sample 50% of your data
df_sampled = df.sample(frac=0.5, random_state=42)

# Redefine X and y based on df_sampled
X = df_sampled[['review', 'above_average_playtime']]
y = df_sampled['review_length_category']

# Split the data into training set and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

In [19]:
```
df_sampled.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 38372 entries, 337302 to 449909
Data columns (total 20 columns):
 #   Column                          Non-Null Count  Dtype   
---  ------                          --------------  -----   
 0   review                          38372 non-null  object  
 1   voted_up                        38372 non-null  object  
 2   votes_up                        38372 non-null  float64 
 3   votes_funny                     38372 non-null  float64 
 4   weighted_vote_score             38372 non-null  float64 
 5   comment_count                   38372 non-null  float64 
 6   steam_purchase                  38372 non-null  object  
 7   received_for_free               38372 non-null  object  
 8   written_during_early_access     38372 non-null  object  
 9   author.num_games_owned          38372 non-null  float64 
 10  author.num_reviews              38372 non-null  float64 
 11  author.playtime_forever         38372 non-null  float64 
 12  author.playtime_last_two_weeks  38372 non-null  float64 
 13  author.playtime_at_review       38372 non-null  float64 
 14  author.last_played              38372 non-null  float64 
 15  timestamp_dev_responded         7 non-null      float64 
 16  developer_response              7 non-null      object  
 17  review_length                   38372 non-null  int64   
 18  review_length_category          38372 non-null  category
 19  above_average_playtime          38372 non-null  int32   
dtypes: category(1), float64(11), int32(1), int64(1), object(6)
memory usage: 5.7+ MB
```

In [20]:
```
# Define pipeline for RandomForest
rf_clf = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier())
])

# Fit the RandomForest model and make predictions
rf_clf.fit(X_train, y_train)
rf_y_pred_train = rf_clf.predict(X_train)
rf_y_pred_test = rf_clf.predict(X_test)

print("Classification Report for Random Forest Classifier (Training Data):")
print(classification_report(y_train, rf_y_pred_train))
print("\nClassification Report for Random Forest Classifier (Test Data):")
print(classification_report(y_test, rf_y_pred_test))
```

```
Classification Report for Random Forest Classifier (Training Data):
              precision    recall  f1-score   support

           0       0.97      0.99      0.98      4331
           1       1.00      0.99      0.99      7553
           2       1.00      1.00      1.00      7343
           3       1.00      1.00      1.00      7633

    accuracy                           0.99     26860
   macro avg       0.99      0.99      0.99     26860
weighted avg       0.99      0.99      0.99     26860


Classification Report for Random Forest Classifier (Test Data):
              precision    recall  f1-score   support

           0       0.68      0.63      0.65      1808
           1       0.65      0.74      0.69      3188
           2       0.78      0.73      0.76      3203
           3       0.93      0.89      0.91      3313

    accuracy                           0.77     11512
   macro avg       0.76      0.75      0.75     11512
weighted avg       0.77      0.77      0.77     11512
```

In [21]:

```
# Define your pipeline for XGBoost
xgb_clf = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', XGBClassifier(use_label_encoder=False))
])

# Fit the XGBoost model and make predictions
xgb_clf.fit(X_train, y_train)
xgb_y_pred_train = xgb_clf.predict(X_train)
xgb_y_pred_test = xgb_clf.predict(X_test)

# Classification report for XGBoost
print("\nClassification Report for XGBoost (Training Data):")
print(classification_report(y_train, xgb_y_pred_train))
print("\nClassification Report for XGBoost (Test Data):")
print(classification_report(y_test, xgb_y_pred_test))
```

```
Classification Report for XGBoost (Training Data):
              precision    recall  f1-score   support

           0       0.73      0.85      0.79      4331
           1       0.80      0.80      0.80      7553
           2       0.93      0.87      0.90      7343
           3       0.99      0.97      0.98      7633

    accuracy                           0.87     26860
   macro avg       0.86      0.87      0.87     26860
weighted avg       0.88      0.87      0.88     26860


Classification Report for XGBoost (Test Data):
              precision    recall  f1-score   support

           0       0.64      0.73      0.68      1808
           1       0.68      0.69      0.69      3188
           2       0.80      0.74      0.77      3203
           3       0.92      0.91      0.92      3313

    accuracy                           0.77     11512
   macro avg       0.76      0.77      0.76     11512
weighted avg       0.78      0.77      0.78     11512
```

It looks like both our models are prone to overfitting on the training data, and doing ok on the test data. We'd like them to do better, so let's tune the hyperparameters of our XGB model (which performed slightly better) using GridSearchCV. Again, we will only use a small subset of the data to speed up processing time.

In [22]:
```python
# Sample a subset of your data for speed
X_train_sampled = X_train.sample(frac=0.1, random_state=42)
y_train_sampled = y_train.sample(frac=0.1, random_state=42)

param_grid = {
    'preprocessor__text__tfidf__max_features': [500, 1000, 2000],
    'classifier__n_estimators': [50, 100, 200],
    'classifier__max_depth': [2, 5, 10],
    'classifier__learning_rate': [0.01, 0.1, 0.2],
}

# Initialize the XGBoost classifier
xgb = XGBClassifier(random_state=42, verbosity=0)

# Initialize GridSearchCV with the XGBoost classifier and parameter grid
grid_search = GridSearchCV(xgb_clf, param_grid, cv=5, verbose=3, n_jobs=-1)

# Fit the GridSearchCV model
grid_search.fit(X_train_sampled, y_train_sampled)

# Get the best parameters found by GridSearchCV
best_params = grid_search.best_params_
print("Best parameters:", best_params)

# Get the best model found by GridSearchCV
best_model = grid_search.best_estimator_

# Predict the training set results using the best model
y_pred_train = best_model.predict(X_train)

# Generate classification report for the training data
report_train = classification_report(y_train, y_pred_train)

# Print the classification report
print("Classification Report (Training Data):\n", report_train)
```

```
Fitting 5 folds for each of 81 candidates, totalling 405 fits
Best parameters: {'classifier__learning_rate': 0.1, 'classifier__max_depth': 10, 'classifier__n
_estimators': 50, 'preprocessor__text__tfidf__max_features': 2000}
Classification Report (Training Data):
               precision    recall  f1-score   support

           0       0.59      0.74      0.65      4331
           1       0.67      0.65      0.66      7553
           2       0.79      0.72      0.76      7343
           3       0.93      0.91      0.92      7633

    accuracy                           0.76     26860
   macro avg       0.75      0.75      0.75     26860
weighted avg       0.77      0.76      0.76     26860
```

Oh no! It looks like our tuning actually led to worse results! Let's just stick with our base model then.

Sentiment Analysis

We are going to use TextBlob's NaiveBayesAnalyzer for our sentiment analysis. The NBA was trained on movie reviews, which is the closes we get to game reviews. To help it out, we are going to provide our model with 4 themes to look for in the data. We want to help our client figure out what it was exactly that people enjoyed about their games. Here are the themes:

In [19]:
```
# Define the themes and their associated words
themes = {
    'music': ['sound', 'music', 'audio', 'instrument', 'soundtrack', 'voice acting', 'song', 'ef
    'story': ['story', 'plot', 'narrative', 'character', 'mission', 'quest', 'writing', 'dialogu
    'game play': ['gameplay', 'rogue-like', 'mechanics', 'controls', 'action', 'fight', 'attack',
    'visuals': ['visuals', 'graphics', 'art', 'images', 'color', 'artwork', 'animation', '2D', '
}
```

Now we want to initiate our analyzer:

In [85]:
```
# Initiate TextBlob's sentiment analyzer
tb = NaiveBayesAnalyzer()

# Define a function to calculate the sentiment scores for each sentence
def get_sentiment(review):
    sentiments = []
    for sentence in review:
        blob = TextBlob(sentence, analyzer=tb)
        sentiment = blob.sentiment.p_pos
        sentiments.append(sentiment)
    return sentiments

# Define a function to calculate the general sentiment score of a review
def get_general_sentiment(review):
    blob = TextBlob(review)
    sentiment = blob.sentiment.polarity
    return sentiment

# Apply general sentiment analysis to each review in the selected data and create a general senti
df_sampled['general_sentiment'] = df_sampled['review'].apply(get_general_sentiment)
```

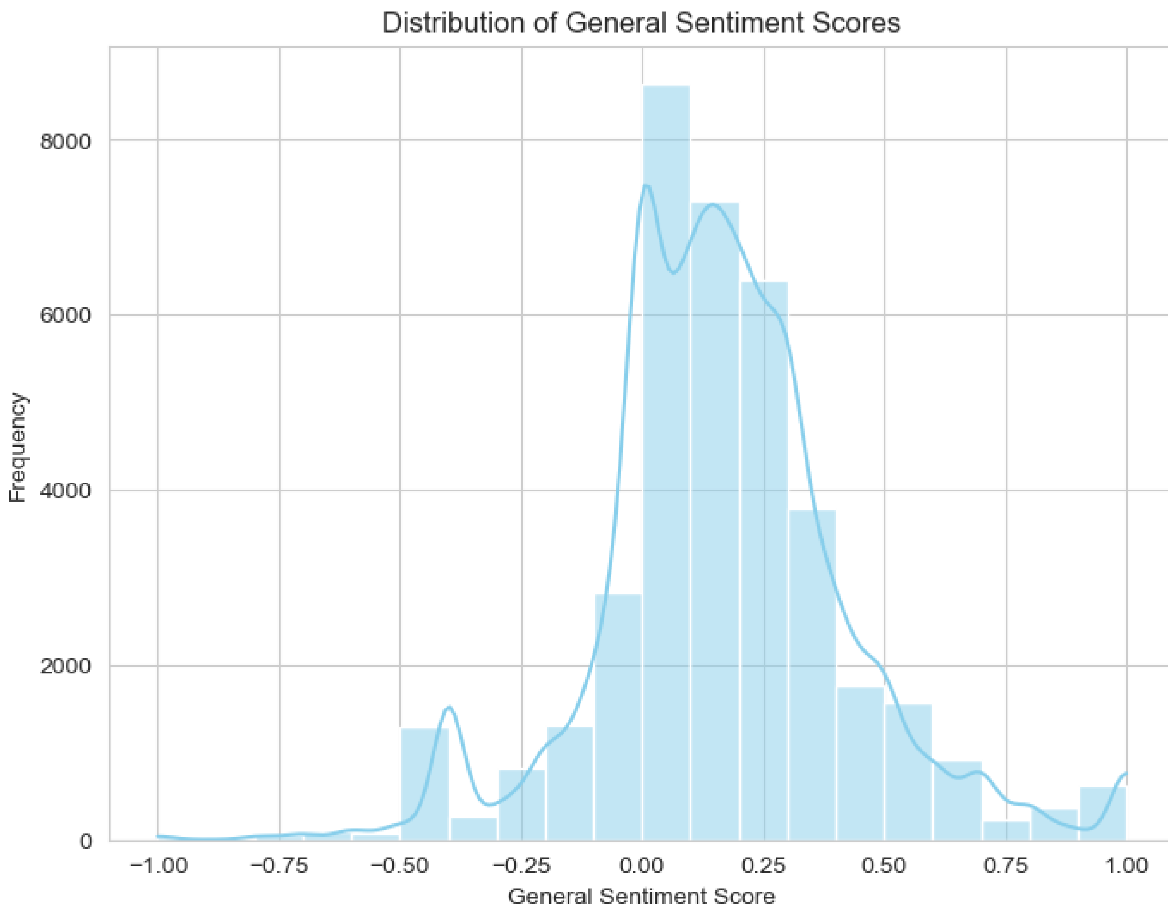
We want to get sentiments on the general review level and on the sentence level. The more fine-grained the better! These functions, get_sentiment(review) and get_general_sentiment(review), takes a review as input and calculates the sentiment scores for each sentence in the review and review at large using TextBlob's sentiment analysis. It returns a list of sentiment scores.

In [102]:
```
# Set up the figure and axes using seaborn
plt.figure(figsize=(8, 6))
sns.set_style("whitegrid")

# Plot the general sentiment scores
sns.histplot(df_sampled['general_sentiment'], bins=20, kde=True, color='skyblue')

# Set labels and title
plt.xlabel('General Sentiment Score')
plt.ylabel('Frequency')
plt.title('Distribution of General Sentiment Scores')

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



This histogram gives us more data than our logistic regression. We can see that rather than a simple binary of recommended or not, players had a range of sentiment concerning what they liked about the game. Now let's try and create a little program that can pick a review at random and display its content, its polarity, and which words within the review are contributing to that polarity based on the themes we provided:

In [100]:
```python
# Select a random review index
review_index = random.randint(0, len(df_sampled) - 1)

# Retrieve the random review
review = df_sampled['review'].iloc[review_index]

theme_polarities = {}
for theme, words in themes.items():
    theme_polarities[theme] = []

    for word in words:
        keyword = f" {word} "  # Add spaces around the keyword to match whole words
        if keyword in review:
            keyword_sentiment = TextBlob(keyword, analyzer=tb).sentiment.p_pos
            theme_polarities[theme].append((word, keyword_sentiment))

# Print random review
print("Review Length:", len(review))
print("Review:", review)

# Print theme words and polarities
print("Theme Words and Polarity")
for theme, polarities in theme_polarities.items():
    print(theme + ":", ", ".join([f"{word}, {polarity}" for word, polarity in polarities]))
```

```
Review Length: 95
Review: go one best game time . 10/10 everything : story , gameplay , difficulty , art , voice
acting .
Theme Words and Polarity
music: voice acting, 0.5484661905425029
story: story, 0.525949953660797
gameplay: gameplay, 0.5
visuals: art, 0.5943396226415094
```

With this review we can see that it was generally positive, and liked the voice acting, story, gameplay, and art, with each recieving over .5 points in positive polarity.

Topic Modeling using LDA

Now for some additional verification, we are going to run an unsupervised learning model to see if it covers similar topics. Specifically we will use Gensim's Latent Dirichlet Allocation (LDA) model. We will prepare the reviews for LDA by removing the stopwords, lemmatizing them, and creating the dictionary and corpus needed for the topic modeling.

```
In [15]: # Define a function to preprocess the texts
def preprocess_text(text):
    # Tokenize the text
    tokens = word_tokenize(text)

    # Remove non-alphabetic tokens, such as punctuation
    words = [token.lower() for token in tokens if token.isalpha()]

    # Filter out stop words
    words = [word for word in words if word not in stop_words]

    # Lemmatize words
    words = [lemmatizer.lemmatize(word) for word in words]

    return words

# Apply preprocessing to the review column
df['tokens'] = df['review'].apply(preprocess_text)

# Tokenize each review string into a list of tokens
tokenized_reviews = list(df['tokens'])

# Create a dictionary representation of the documents
dictionary = Dictionary(tokenized_reviews)

# Create Bag-of-words representation of the documents
corpus = [dictionary.doc2bow(review) for review in tokenized_reviews]

# print out the first 5 documents in the corpus
for doc in corpus[:5]:
    print([[dictionary[id], freq] for id, freq in doc])
```

```
[['acting', 1], ['art', 1], ['beautiful', 1], ['fun', 1], ['game', 1], ['gameplay', 1], ['grea
t', 1], ['like', 1], ['music', 1], ['really', 1], ['supergiant', 1], ['voice', 1]]
[['acting', 1], ['art', 1], ['fun', 1], ['game', 4], ['like', 2], ['really', 2], ['voice', 1],
['amazing', 1], ['animation', 1], ['annoying', 1], ['aspect', 1], ['aswell', 1], ['bastion',
1], ['beat', 1], ['becomes', 2], ['button', 2], ['combat', 1], ['death', 2], ['decide', 1], ['d
eep', 1], ['design', 1], ['disappointed', 1], ['escape', 1], ['everything', 1], ['extremely',
1], ['fan', 1], ['feel', 3], ['first', 1], ['get', 1], ['going', 1], ['grindy', 1], ['hades',
2], ['hour', 1], ['however', 1], ['issac', 1], ['least', 1], ['loose', 1], ['lot', 1], ['love',
1], ['made', 1], ['main', 1], ['mashy', 1], ['massive', 1], ['mid', 1], ['motivation', 1], ['ov
erall', 1], ['polished', 1], ['press', 1], ['punishing', 2], ['quite', 1], ['recommend', 1],
['recommendation', 1], ['repetitive', 1], ['replaying', 1], ['rogue', 2], ['sale', 1], ['soundt
rack', 1], ['still', 1], ['story', 2], ['tedious', 1], ['thumb', 1], ['trying', 1], ['underworl
d', 1], ['upgrade', 1], ['upgraded', 1], ['way', 1], ['weapon', 1], ['would', 2]]
[['art', 1], ['beautiful', 1], ['fun', 1], ['weapon', 1], ['loop', 1], ['perfect', 1]]
[['music', 1], ['combat', 1], ['story', 1], ['upgrade', 1], ['althought', 1], ['best', 1], ['da
ddy', 1], ['difficulty', 2], ['making', 1], ['many', 1], ['market', 1], ['permanent', 1], ['pla
y', 1], ['powered', 1], ['replayabilty', 1], ['tweaking', 1], ['writting', 1]]
[['fun', 1], ['alot', 1], ['die', 1], ['lol', 1], ['u', 1]]
```

In [17]:
```
# Define the number of topics for the LDA model
num_topics = 10

# Train the LDA model
lda_model = LdaModel(corpus, num_topics=num_topics, id2word=dictionary)

# Get the top 10 topics in the LDA model
top_topics = lda_model.show_topics(num_topics=10, num_words=10)

# Print the top 10 topics as single words
for topic_id, topic in top_topics:
    topic_words = [word.split('*')[1].replace('"', '').strip() for word in topic.split('+')]
    topic_words = ', '.join(topic_words)
    print(f"Topic {topic_id + 1}: {topic_words}")
```

```
Topic 1: alive, step, stay, arpg, gay, door, awhile, kick, status, v
Topic 2: game, run, weapon, feel, make, hades, character, different, get, like
Topic 3: hot, e, goty, de, que, fuck, everyone, dating, gone, f
Topic 4: game, hour, time, play, early, still, playing, access, get, even
Topic 5: game, great, story, gameplay, art, character, voice, fun, music, combat
Topic 6: game, good, access, like, early, story, get, bad, even, time
Topic 7: pet, dash, stab, hell, cerberus, game, get, yes, dog, love
Topic 8: greek, mythology, hades, god, die, underworld, slash, hack, zagreus, escape
Topic 9: good, game, like, fun, really, rogue, dead, cell, play, lot
Topic 10: game, best, one, played, supergiant, ever, roguelike, year, amazing, like
```

```
It's hard to get a clear theme from these. Lots of action words, so perhaps 'gameplay' is a
good theme? Or perhaps its too general. Let's check the top bigrams to see if they reveal
anything else about the review topics:
```

In [72]:
```
# Initialize the bigram model
bigram_model = Phrases(tokenized_reviews, min_count=5, threshold=100)

# Get the top bigrams
top_bigrams = list(bigram_model.export_phrases())

# Print the top 10 bigrams
print("Top 10 bigrams:")
for bigram in top_bigrams[:10]:
    print(bigram)
```

```
Top 10 bigrams:
button_mashy
hack_slash
learning_curve
keyboard_mouse
greek_mythology
gon_na
early_access
floating_head
fishing_minigame
top_notch
```

Some of these look helpful. We might categorize button_mashy, hack_slash, learning_curve, keyboard_mouse, and fishing_minigame as 'gameplay' topics, and greek_mythology as 'story.' Let's see if we get any more clarity by limiting our bigrams to our pre-selected themes:

In [79]:
```python
# Create a dictionary to store the theme bigrams
theme_bigrams = {}

# Filter the top bigrams based on themes and their synonyms
for theme, words in themes.items():
    theme_bigrams.setdefault(theme, [])

    for bigram in top_bigrams:
        if any(word in bigram for word in words):
            theme_bigrams[theme].append(''.join(bigram))

# Print the top 5 bigrams for each theme
for theme, bigrams in theme_bigrams.items():
    print(f"Top 5 bigrams for {theme.capitalize()} theme:")
    for bigram in bigrams[:5]:
        count = len(bigram.replace('_', ''))
        print(f"{bigram}: Count - {count}")
    print()
```

```
Top 5 bigrams for Music theme:
sound_track: Count - 10
instead_audio: Count - 12
audio_eargasm: Count - 12
musical_score: Count - 12
mass_effect: Count - 10

Top 5 bigrams for Story theme:
side_quest: Count - 9
question_asked: Count - 13
family_drama: Count - 11
extended_family: Count - 14
answer_question: Count - 14

Top 5 bigrams for Gameplay theme:
attack_pattern: Count - 13
power_ups: Count - 8
el_combate: Count - 9
power_creep: Count - 10
micro_transaction: Count - 16

Top 5 bigrams for Visuals theme:
late_party: Count - 9
add_cart: Count - 7
vibrant_color: Count - 12
color_palette: Count - 12
farewell_earthly: Count - 15
```

That is definitely more useful! We we are able to see which of the words are associated with each them, and how often those pairs appeared. Now let's step back and see how often our themes appeared more generally.

In [35]:

```
# Create a dictionary to store theme appearance counts
theme_appearance_counts = {theme: 0 for theme in themes}

# Iterate over each review
for review in df['review']:
    # Check if each theme is mentioned in the review at least once
    for theme, words in themes.items():
        if any(word in review for word in words):
            theme_appearance_counts[theme] += 1

# Print the theme appearance counts
for theme, count in theme_appearance_counts.items():
    print(f"{theme}: {count} appearances")
```

```
music: 15901 appearances
story: 28991 appearances
gameplay: 25387 appearances
visuals: 17288 appearances
```

In [36]:

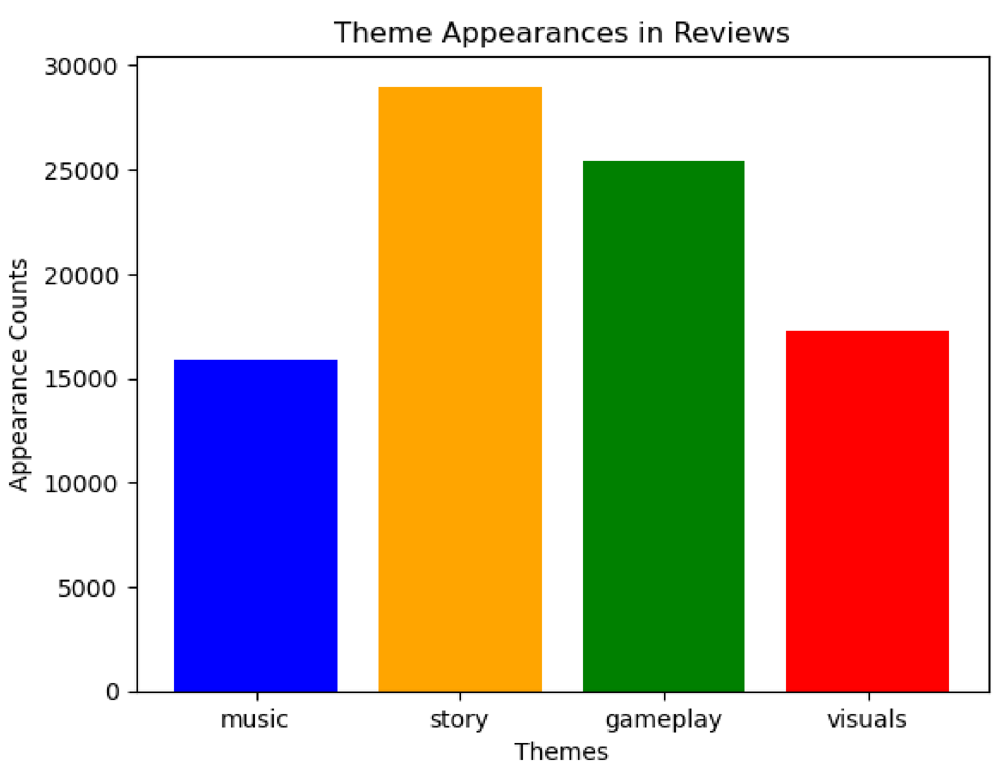
```
# Define the themes and their appearance counts
themes = ['music', 'story', 'gameplay', 'visuals']
appearance_counts = [15901, 28991, 25387, 17288]

# Define colors for the bars
colors = ['blue', 'orange', 'green', 'red']

# Create a bar chart with colored bars
plt.bar(themes, appearance_counts, color=colors)

# Customize the chart
plt.xlabel('Themes')
plt.ylabel('Appearance Counts')
plt.title('Theme Appearances in Reviews')

# Display the chart
plt.show()
```



Conclusion

1. The reviews for the game Hades generally expressed positive sentiment, although the overall level of positivity falls within the range of 0 to 0.25.
2. When discussing their experiences with the game, players frequently emphasized the importance of the game's story. This indicates that the narrative elements of Hades are a significant aspect of player enjoyment.
3. It appears that players may have limited vocabulary when describing their appreciation for the 'music' and 'visuals' in Hades. This suggests that while players find these aspects appealing, they may struggle to articulate their specific likes or preferences regarding the music and visual elements of the game.

Reccomendations

Based on these findings, I would recommend SuperGiant Games to continue focusing on the strong storytelling elements of Hades, as players consistently highlighted this aspect. Additionally, efforts can be made to enhance players' ability to express their positive impressions of the 'music' and 'visuals' by potentially providing prompts or specific questions related to these aspects in reviews or feedback forms. This would help gather more detailed and insightful feedback on the game's audio and visual components.

Limitations

Given the computational limitations, making confident predictions about the specific aspects of the game that received positive reviews remains challenging. However, we were successful in adding complexity to the analysis of reviews by incorporating sentiment analysis and exploring themes within the text. This approach has revealed potential insights and indicates the value of delving deeper into the analysis. Further investigation into the sentiment scores of specific themes and their impact on overall sentiment could provide valuable insights into the aspects of the game that resonate with reviewers. Despite the challenges, our findings suggest that there is merit in continuing to explore and refine our analysis methods to gain a deeper understanding of the factors contributing to positive reviews.

For Further Research

I'd like to check the sentiment scores for each of our themes. So I need code that looks at the sentiment scores of the sentences of each review, determines whether or not the sentence is referring to a particular one of our 4 themes, and then adds that score to the proper theme column. For each review.

In [ ]:
```
# Create theme sentiment columns in the DataFrame
for theme in themes:
    df[theme + '_sentiment'] = 0.0

# Iterate over each review
for review in df['review']:
    # Initialize sentiment scores for each theme
    theme_scores = {theme: 0.0 for theme in themes}

    # Calculate sentiment score for each sentence in the review
    for sentence in review:
        for theme, words in themes.items():
            if any(word in sentence for word in words):
                sentiment = TextBlob(sentence, analyzer=tb).sentiment.p_pos
                theme_scores[theme] += sentiment

    # Add the sentiment scores to the DataFrame
    for theme, score in theme_scores.items():
        df.loc[df['review'] == review, theme + '_sentiment'] = score
```

In [ ]:
```
# Apply sentiment analysis to each sentence in the selected data
df_sampled['sentiment'] = df_sampled['review'].apply(get_sentiment)
# Create theme-specific sentiment score columns
for theme in themes:
    theme_column = f'{theme}_sentiment'
    df_sampled[theme_column] = df_sampled['sentiment'].apply(lambda sentiments: [sentiment for se
```

In [ ]:
```
# Set the size of the scatter points
point_size = 50

# Create a scatter plot for each theme
fig, axes = plt.subplots(nrows=len(themes), figsize=(8, 12))

for i, (theme, ax) in enumerate(zip(themes, axes)):
    sentiment_column = f'{theme}_sentiment'

    # Get the sentiment scores and review lengths for the theme
    sentiment_scores = df[sentiment_column].explode().values
    review_lengths = df['review'].apply(len).values

    # Create the color map for sentiment scores
    cmap = plt.cm.coolwarm
    norm = plt.Normalize(vmin=min(sentiment_scores), vmax=max(sentiment_scores))
    colors = cmap(norm(sentiment_scores))

    # Create the scatter plot
    ax.scatter(review_lengths, sentiment_scores, c=colors, cmap='coolwarm', s=point_size, alpha=(

    ax.set_xlabel('Review Length')
    ax.set_ylabel('Sentiment Score')
    ax.set_title(f'Sentiment Scores vs Review Length for {theme.capitalize()} Theme')
    ax.legend()

plt.tight_layout()
plt.show()
```

In [ ]:

```
# Set up colors for each theme
theme_colors = ['red', 'blue', 'green', 'orange']

# Set the width of each bar
bar_width = 0.15

# Set the x coordinates for the bars
x = np.arange(len(themes))

# Plot the sentiment scores for each theme side by side
plt.figure(figsize=(8, 6))

for i, theme in enumerate(themes.keys()):
    sentiment_column = f'{theme}_sentiment'
    theme_sentiments = df[sentiment_column].explode().dropna()

    # Calculate the x position for each theme's bar
    x_pos = x[i]

    # Plot histogram of sentiment scores with the corresponding color and x position
    plt.hist(theme_sentiments, bins=5, range=(0, 1), alpha=0.7, edgecolor='black',
             color=theme_colors[i], label=theme, align='mid', rwidth=bar_width)

plt.xlabel('Sentiment Score')
plt.ylabel('Frequency')
plt.title('Sentiment Distribution for Themes')
plt.xticks(x, themes.keys())
plt.legend()
plt.tight_layout()
plt.show()
```

In [ ]:
```
# Create a dictionary to store theme appearance counts
theme_appearance_counts = {theme: 0 for theme in themes}

# Define the threshold for selecting bigrams
threshold = 5

# Iterate over each review
for review in df['review']:
    # Check if each theme is mentioned in the review at least once
    for theme, words in themes.items():
        if any(word in review for word in words):
            theme_appearance_counts[theme] += 1

            # Create a list of theme-related sentences
            theme_sentences = [sentence for sentence in review.split('.') if any(word in sentenc

            # Tokenize the theme-related sentences
            tokenized_sentences = [word_tokenize(sentence.lower()) for sentence in theme_sentenc

            # Create a finder to identify bigrams
            finder = BigramCollocationFinder.from_documents(tokenized_sentences)

            # Apply a frequency filter to select relevant bigrams
            finder.apply_freq_filter(threshold)

            # Get the top 5 most common bigrams with sentiment words
            top_bigrams = finder.nbest(BigramAssocMeasures.raw_freq, 5)

            # Print the top bigrams
            print(f'Top bigrams for {theme.capitalize()} theme:')
            for bigram in top_bigrams:
                print(' '.join(bigram))
            print()

# Print the theme appearance counts
for theme, count in theme_appearance_counts.items():
    print(f"{theme}: {count} appearances")
```

Maybe check to see how my pre-selected themes did in terms of meaningful score using the LDA:

In [ ]:
```
# Create a dictionary to store theme sentiment scores
theme_sentiments = {theme: [] for theme in themes}

# Iterate over each review
for review in df['review']:
    # Calculate sentiment score for each sentence in the review
    for sentence in review:
        for theme in themes:
            if any(word in sentence for word in themes[theme]):
                sentiment = TextBlob(sentence, analyzer=tb).sentiment.p_pos
                theme_sentiments[theme].append(sentiment)

# Print theme sentiment scores
for theme, sentiments in theme_sentiments.items():
    print(f"{theme.capitalize()} Sentiment Scores: {sentiments}")
```