ML Recommendations

This notebook implements and evaluates the two different recommendation systems based on the recommendations we found in the EDA.ipynb.

In summary, those are

Model 1: TF-IDF + Cosine Similarity (Content-based)

Focuses purely on similarity of game descriptions ('About the game')

Model 2: K-Nearest Neighbors (KNN) on Combined Features (Hybrid)

Considers text, numerical and categorical features to find similar games

(More detailed descripion in EDA.ipynb)

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import MinMaxScaler # Choose one for scaling
from scipy.sparse import hstack, csr_matrix
```

In [2]:
```
df = pd.read_csv("Dataset/games_eda.csv")

print("Sample names:", df['Name'].sample().tolist())
```

```
Sample names: ['Sledders']
```

Create a mapping from game name to index for easy lookup

In [3]:
```
df_unique_names = df.drop_duplicates(subset=['Name'], keep='first') # Handle pot
indices = pd.Series(df_unique_names.index, index=df_unique_names['Name'])

print(f"Number of unique game names: {len(indices)}")
```

```
Number of unique game names: 22066
```

Model 1: TF-IDF + Cosine Similarity

Select text from the game descriptions in 'About the game' and handle missing values with empty strings to ensure vectorizer can process all rows.

In [4]:
```
corpus = df['About the game'].fillna('') # Fill NaN values with empty strings
```

We use the TfidfVectorizer from scikit-learn to convert the raw text descriptions into matrix of TTF-IDF features

What is TF-IDF (Term Frequency-Inverse Document Frequency)?

It's a technique to reflect how important a word is to a document in a collection. It increases proportionally to the number of time it appears in a document but offsetted by the frequence of the word in the corpus.

```
In [5]: # Use stop_words='english' to remove common English words
        # max_df=0.8 -> ignore terms that appear in more than 80% of the documents
        # min_df=5   -> ignore terms that appear in less than 5 documents
        # ngram_range=(1, 2) -> consider both single words and two-word phrases
        tfidf_vectorizer = TfidfVectorizer(stop_words='english', max_df=0.8, min_df=5, n
        tfidf_matrix = tfidf_vectorizer.fit_transform(corpus)


        print(f"TF-IDF matrix shape: {tfidf_matrix.shape}") # (number of games, number o
```

```
TF-IDF matrix shape: (22105, 89540)
```

Computes the pairwise similarity between all games based on their TF-IDF vectors.

What is Cosine Simlarity?

It's the cosine of the angle between two non-zero vectors. Its highly effective for texts due to the high dimensions (TF-IDF results) and focuses on the orientation (content) rather than the magnitude (length of description).

```
In [6]: cosine_sim_matrix = cosine_similarity(tfidf_matrix, tfidf_matrix) # Compute cosi
        print(f"Cosine similarity matrix shape: {cosine_sim_matrix.shape}")
```

```
Cosine similarity matrix shape: (22105, 22105)
```

The TFIDF function

Function to take in a game title as input and return 10 most similar games based on cosine similarity.

```
In [7]: def get_tfidf_recommendations(title, cosine_sim=cosine_sim_matrix, data_indices=
            """
            Gets game recommendations based on TF-IDF cosine similarity.

            Args:
                title (str): The title of the game to find recommendations for.
                cosine_sim (np.ndarray): The precomputed cosine similarity matrix.
                data_indices (pd.Series): Series mapping game titles to their index.
                data_df (pd.DataFrame): The original dataframe to get game names from in

            Returns:
                list: A list of recommended game titles, or None if title not found.
            """
            if title not in data_indices:
                print(f"Error: Game '{title}' not found in the dataset.")
                # Try finding partial matches
                possible_matches = [name for name in data_indices.index if title.lower()
                print(f"Finding similar games for: {possible_matches[0]} instead as '{ti
                title = possible_matches[0] if possible_matches else None
                if not title:
                    return None

            idx = data_indices[title] # Get the index of the game in the DataFrame
```

```
    # Get the pairwise similarity scores of all games with that game
    # enumerate adds a counter to the iterable, list converts it to a list of (i
    sim_scores = list(enumerate(cosine_sim[idx]))

    # Sort the games based on the similarity scores (descending order)
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)

    # Get the scores of the 10 most similar games (excluding the game itself, wh
    sim_scores = sim_scores[1:11]

    # Get the game indices from the (index, score) tuples
    game_indices = [i[0] for i in sim_scores]

    # Return the titles of the top 10 most similar games
    return [(data_df['Name'].iloc[i], score) for i, score in sim_scores]
```

Model 2: K-Nearest Neighbors (KNN) on Combined Features

In [8]:
```
tfidf_features = tfidf_matrix # Reuse the TF-IDF matrix as features for KNN
```

Numerical Features (Potentially relevant Features like popularity, rating, age, playtime)

In [9]:
```
numerical_cols = [
    'Log Peak CCU', 'Review Ratio', 'Game Age (Days)',
    'Log Median playtime forever', 'Log Achievements', 'Log DLC count',
    'Num Languages'
]
```

Handle NaN values in case still have

In [10]:
```
df[numerical_cols] = df[numerical_cols].fillna(0).replace([np.inf, -np.inf], 0)
numerical_features_raw = df[numerical_cols].values
```

Categorical Features (One-Hot Encoded Tags, Genres, Categories)

In [11]:
```
tag_cols = [col for col in df.columns if col.startswith('Tags_')]
genre_cols = [col for col in df.columns if col.startswith('Genres_')]
category_cols = [col for col in df.columns if col.startswith('Categories_')]

# Combine all one-hot encoded features
categorical_features = df[tag_cols + genre_cols + category_cols].values
```

Convert dense categorical features to sparse matrix format for efficient combination

What is Spare matrix?

Spare matrix store only non-zero values, to save memory and speed up computations

In [12]:
```
categorical_features_sparse = csr_matrix(categorical_features)
```

Training the KNN model

We initialize and fit a NearestNeighbors model from scikit-learn.

StandardScaler: Scales to zero mean and unit variance (good if data is normally distributed)

MinMaxScaler: Scales to a range [0, 1] (good if data isn't normally distributed or for distance metrics)

Since we found that our data was very right-skewed, we use MinMaxScaler here

In [13]:
```
scaler = MinMaxScaler()
numerical_features_scaled = scaler.fit_transform(numerical_features_raw)
```

Convert scaled numerical features to sparse matrix format again

In [14]:
```
numerical_features_scaled_sparse = csr_matrix(numerical_features_scaled)
```

Combine TF-IDF, numerical, and categorical features into a single sparse matrix

In [15]:
```
combined_features = hstack([
    tfidf_features,
    numerical_features_scaled_sparse,
    categorical_features_sparse
])

print(f"Combined features matrix shape: {combined_features.shape}")
print(f"Feature breakdown: TF-IDF ({tfidf_features.shape[1]}), Scaled Numerical
```

```
Combined features matrix shape: (22105, 90063)
Feature breakdown: TF-IDF (89540), Scaled Numerical (7), One-Hot Categorical (51
6)
```

We want 11 neighbors (1 self + 10 recommendations)

'cosine' metric is often good for high-dimensional, sparse data like this

'brute' algorithm checks all points, suitable for sparse data

In [16]:
```
knn_model = NearestNeighbors(n_neighbors=11, metric='cosine', algorithm='brute',
knn_model.fit(combined_features)
```

Out[16]:
```
▾                        NearestNeighbors                        i ?

NearestNeighbors(algorithm='brute', metric='cosine', n_jobs=-1, n_neigh
bors=11)
```

The KNN Function

In [ ]:
```
def get_knn_recommendations(title, model=knn_model, features=combined_features,
    """
    Gets game recommendations based on KNN on combined features.

    Args:
        title (str): The name of the game to get recommendations for.
        model (NearestNeighbors): The fitted KNN model.
        features (csr_matrix): The combined feature matrix used for fitting KNN.
        data_indices (pd.Series): Series mapping game titles to their index.
        data_df (pd.DataFrame): The original dataframe to get game names from in

    Returns:
        list: A list of recommended game titles, or None if title not found.
    """
    if title not in data_indices:
        print(f"Error: Game '{title}' not found in the dataset.")
        # Try finding partial matches
        possible_matches = [name for name in data_indices.index if title.lower()
        print(f"Finding similar games for: {possible_matches[0]} instead as '{ti
        title = possible_matches[0] if possible_matches else None
        if not title:
            return None

    # Get the index of the game
    idx = data_indices[title]

    # Get the feature vector for the target game
    query_vector = features[idx]

    # Find the nearest neighbors
    # distances: distance values of neighbors
    # indices_knn: indices of neighbors in the original dataset
    distances, indices_knn = model.kneighbors(query_vector)


    # The first index (indices_knn[0][0]) is the game itself. Exclude it.
    neighbor_indices = indices_knn[0][1:]
    neighbor_distances = distances[0][1:]

    # Return the names of the recommended games and their similarity scores
    return [(data_df['Name'].iloc[i], 1 - d)  # 1-distance → "cosine-like" simil
            for i, d in zip(neighbor_indices, neighbor_distances)]
```

Main Loop to try out the model

In [22]:
```
target_game = 'Terraria'

print(f"\n--- Recommendations for '{target_game}' ---")

# Model 1: TF-IDF + Cosine Similarity
print("\nModel 1 (TF-IDF + Cosine Similarity) Recommendations:")
for rank, (name, score) in enumerate(get_tfidf_recommendations(target_game), 1):
    print(f"{rank:2}. {name:50}  ({score:.3f})")

# Model 2: KNN on Combined Features
print("\nModel 2 (KNN on Combined Features) Recommendations:")
for rank, (name, score) in enumerate(get_knn_recommendations(target_game), 1):
    print(f"{rank:2}. {name:50}  ({score:.3f})")
```

```
--- Recommendations for 'Terraria' ---

Model 1 (TF-IDF + Cosine Similarity) Recommendations:
 1. STERN                                                (0.159)
 2. Vagabond                                             (0.124)
 3. Realmcraft VR                                        (0.097)
 4. Darkout                                              (0.097)
 5. Forsaken Isle                                        (0.093)
 6. Fractured Online                                     (0.085)
 7. Wizards and Warlords                                 (0.085)
 8. The Mole Men                                         (0.083)
 9. Drive 4 Survival                                     (0.081)
10. The Sandbox Evolution - Craft a 2D Pixel Universe!  (0.080)

Model 2 (KNN on Combined Features) Recommendations:
 1. Niffelheim                                           (0.789)
 2. Starbound                                            (0.785)
 3. Crea                                                 (0.774)
 4. Conan Exiles                                         (0.762)
 5. Necesse                                              (0.757)
 6. Tinkertown                                           (0.756)
 7. Dig or Die                                           (0.749)
 8. Valheim                                              (0.735)
 9. Signs of Life                                        (0.734)
10. DRAGON QUEST BUILDERS™ 2                             (0.731)
```

Evaluation

--- Recommendations for 'Terraria' ---

Model 1 (TF-IDF + Cosine Similarity) Recommendations:

1. STERN (0.159)
2. Vagabond (0.124)
3. Realmcraft VR (0.097)
4. Darkout (0.097)
5. Forsaken Isle (0.093)
6. Fractured Online (0.085)
7. Wizards and Warlords (0.085)
8. The Mole Men (0.083)
9. Drive 4 Survival (0.081)
10. The Sandbox Evolution - Craft a 2D Pixel Universe! (0.080)

Model 2 (KNN on Combined Features) Recommendations:

1. Niffelheim (0.789)
2. Starbound (0.785)
3. Crea (0.774)
4. Conan Exiles (0.762)
5. Necesse (0.757)
6. Tinkertown (0.756)
7. Dig or Die (0.749)
8. Valheim (0.735)

9. Signs of Life (0.734)
10. DRAGON QUEST BUILDERS™ 2 (0.731)

Model 1: TF-IDF + Cosine Similarity (Description)

Low Similarity scores, in cosine similarity, 1 represents identical. In our model, the model actual return quite low values. Indicating that purely based on text descriptions processed by TF-IDF, none of the games were highly similar to Terraria.

Although not to put it off, there is potential for some relation, like the "Sandbox Evolution" game hints at a sandbox style game, and "Darkout" is survival/exploration game that is similar to Terraria.

Overall, we find that the model struggles to find strong matches based solely on "About the game" text. The core appeal may be better captured in the tags, genres and gameplay mechanics which aren't fully represented by the description, or similar games don't use overlapping categories for TF-IDF to score them highly. The recommendations become less focused due to this and potentially include more obscure, less related titles.

Model 2: KNN on Combined Features (Text, metadata, categories)

High similarity scores, unlike the first model, the scores were quite high, ranging 0.731 to 0.789 for terraria, showing higher degree of similarity based on the combined features.

The list of games were also quite relevant overall

Starbound: Often called terraria in space

Crea, Necesse, Tinkertown, Dig or Die, Signs of Life: 2D sandbox around crafting, survival, exploration

Valheim, Conan Exiles: Is 3D but highly popular survival/crafting game around gathering, building, fighting and exploring.

DRAGON QUEST BUILDERS™ 2: Block RPG with crafting and sandbox elements

Overall, the model provided quite highly relevant recommendations. The inclusion of numerical features (like popularity, playtime), categorical (tags, genres) and text descriptions allowed the KNN to identify games similar in playstyle, genre and potentially player reception, even if the text descriptions don't perfectly describe the game.

Conclusion

Terraria is a game strongly tied to the genre (sandbox, survival, crafting) and core mechanics on top of just descriptions. Model 2 provided a significantly more relevant and intuitive recommendations based on this info.

Low scores in Model 1 suggest text descriptions often miss games that are functionally similar but described differently. Model 2 was more successfully in this case due to deploying a hybrid approach, and making use of the categorical tags/genres and numerical metadata to capture the aspects of similarity that the text alone don't. The features identified in EDA (tags, genres, popularity metrics) seemed crucial in finding good recommendations for this problem.