Level Up Your Gaming Experience: A Steam Game Recommendation Adventure

Theodor Östling & Emil Wallberg

August 28, 2025

1 Introduction

Welcome to the world of endless gaming possibilities! In this ever-expanding universe of digital entertainment, finding the perfect game can be quite a quest. With millions of games to choose from, it's like navigating a galaxy of options. Fear not, fellow gamers! We're here to introduce you to our solution – a Steam Game Recommendation System.

2 Background

Steam, a colossal platform in the gaming realm, has become the go-to destination for gamers worldwide. Each player's journey is encapsulated within their unique digital profile, a testament to the myriad adventures and conquests in the gaming universe. In our pursuit of reshaping the landscape of game recommendations, we draw inspiration from a Kaggle dataset that captures the essence of Steam profiles and games. If you're curious to explore the dataset further, you can find it at in section 8.

To fully grasp the concepts discussed in this blog post, it's essential to have a basic understanding of two machine learning concepts: K-Nearest Neighbors (K-NN) models and cosine similarity. Let's embark on a brief exploration of these concepts.

2.1 K-Nearest Neighbors (K-NN)

The K-Nearest Neighbors algorithm is a powerful and intuitive machine learning model commonly used in recommendation systems. The core idea behind K-NN is to predict the outcome of data points based on the majority class of its k-nearest neighbors. In the context of our game recommender, each game serves as a data point, and the "neighbors" are other gamers with similar user interactions.

However, it's important to note that K-NN may struggle with cold start scenarios, where there are new games or users with no interaction history.

2.2 Cosine Similarity

Cosine similarity is a metric used to measure the similarity between two vectors, commonly employed in recommendation systems. In our context, these vectors represent user profiles or game interactions. The cosine similarity between two vectors is calculated by taking the dot product of the vectors and dividing it by the product of their magnitudes.

For our K-NN model:

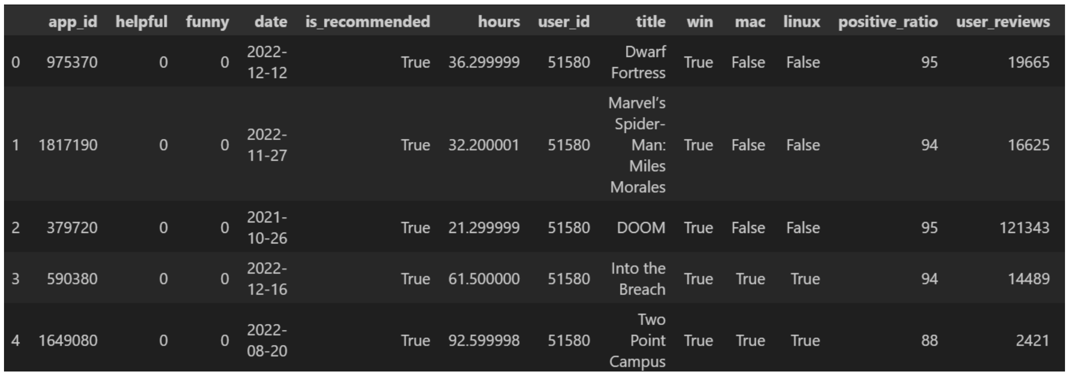
Dot Product: Measures the alignment of two vectors. Magnitude: Represents the "length" or strength of a vector.

The resulting cosine similarity score ranges from -1 (completely dissimilar) to 1 (completely similar). A score closer to 1 indicates higher similarity.

Understanding these concepts lays the foundation for unraveling the magic behind our Steam game recommendation system. Now, let's delve into the nitty-gritty details of the implementation and witness how these concepts come to life in our code.

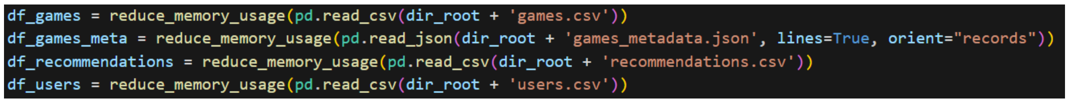
3 Database

Let's talk data! The database we have chosen contains 3 different sets, games, recommendations and users. The dataset contains over 41 million cleaned and preprocessed user recommendations, which was a problem computation wise. For this problem we first of all made a function WHO's purpose was to reduce the memory by changing from *float64* to *float34*. Furthermore, we choose to drop some unnecessary columns for data preprocessing, see figure 4. Sadly this did not make our memory problem disappear, which is why we choose to only use the first 300000 items in our data when testing, see Listing 1. It is also worth mentioning that we merged our 3 different sets into one set called *Data*. We did this by first merging recommendations and games based on their *appid*, which means that the rows with the same *appid* are combined. After this we combined the resulting data with our user user data, but here we combined based on *userid* instead, see figure 1.



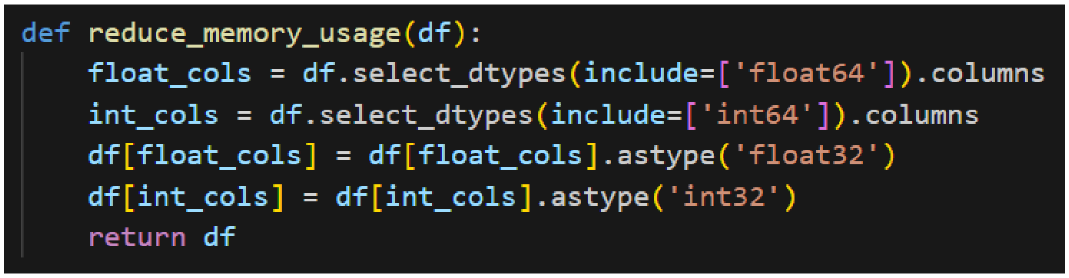
	app_id	helpful	funny	date	is_recommended	hours	user_id	title	win	mac	linux	positive_ratio	user_reviews
0	975370	0	0	2022-12-12	True	36.299999	51580	Dwarf Fortress	True	False	False	95	19665
1	1817190	0	0	2022-11-27	True	32.200001	51580	Marvel's Spider-Man: Miles Morales	True	False	False	94	16625
2	379720	0	0	2021-10-26	True	21.299999	51580	DOOM	True	False	False	95	121343
3	590380	0	0	2022-12-16	True	61.500000	51580	Into the Breach	True	True	True	94	14489
4	1649080	0	0	2022-08-20	True	92.599998	51580	Two Point Campus	True	True	True	88	2421

Figure 1: A chunk of our data file *Data*



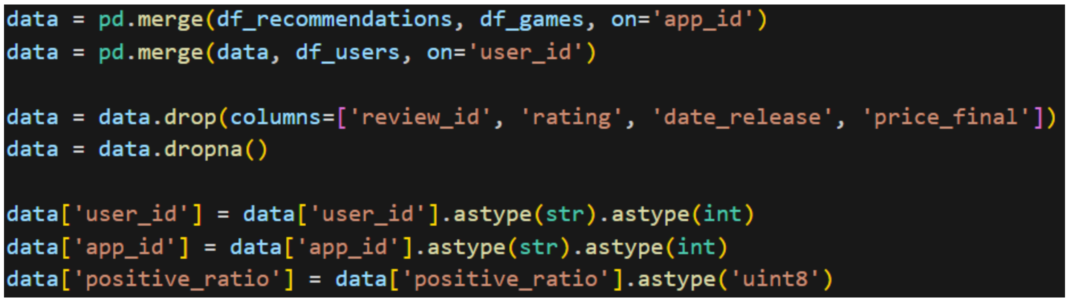
```
df_games = reduce_memory_usage(pd.read_csv(dir_root + 'games.csv'))
df_games_meta = reduce_memory_usage(pd.read_json(dir_root + 'games_metadata.json', lines=True, orient="records"))
df_recommendations = reduce_memory_usage(pd.read_csv(dir_root + 'recommendations.csv'))
df_users = reduce_memory_usage(pd.read_csv(dir_root + 'users.csv'))
```

Figure 2: Our Data



```
def reduce_memory_usage(df):
    float_cols = df.select_dtypes(include=['float64']).columns
    int_cols = df.select_dtypes(include=['int64']).columns
    df[float_cols] = df[float_cols].astype('float32')
    df[int_cols] = df[int_cols].astype('int32')
    return df
```

Figure 3: Reduce Memory Function



```
data = pd.merge(df_recommendations, df_games, on='app_id')
data = pd.merge(data, df_users, on='user_id')

data = data.drop(columns=['review_id', 'rating', 'date_release', 'price_final'])
data = data.dropna()

data['user_id'] = data['user_id'].astype(str).astype(int)
data['app_id'] = data['app_id'].astype(str).astype(int)
data['positive_ratio'] = data['positive_ratio'].astype('uint8')
```

Figure 4: Data prepossessing

4 Implementation

4.1 Foundation: User-Game Interaction Matrix

To kick off our journey, we begin by constructing a matrix that encapsulates the very essence of user interactions with games. Each row represents a unique gamer, each column a game title, and every cell a reflection of the user's sentiment or engagement with a specific game.

```
user_game_matrix = data.head(300000).pivot_table(
index='user_id', columns='title', values='positive_ratio', fill_value=0
)
```

Listing 1: Pivot

This matrix is a treasure trove of insights, mapping out the gaming preferences of users in a structured and comprehensible manner. Sparse Matrix Conversion for Efficiency

To ensure optimal memory usage and computational efficiency, we convert our user-game interaction matrix into a sparse matrix.

```
sparse_user_game_matrix = csr_matrix(user_game_matrix.values)
```

Listing 2: Creating sparse matrix

This step is crucial, especially when dealing with large datasets, as it streamlines the computational process without compromising on the richness of the data.

4.2 Building Bridges with K-Nearest Neighbors (K-NN)

The heart of our recommendation system lies in the implementation of a K-Nearest Neighbors (K-NN) model. This model works on the principle of identifying users with similar gaming profiles, guiding our users towards games that align with their preferences.

```
knn_model = NearestNeighbors(metric='cosine', algorithm='brute')
knn_model.fit(user_game_matrix.values)
```

Listing 3: Building KNN model

By leveraging the cosine similarity metric, the model pinpoints the nearest neighbors for each user, paving the way for personalized game recommendations.

4.3 The Recommendation Function Unveiled

Let's dive into the recommendation function, where the magic truly happens.

```python
def recommendations(user_id):
    distances, indices = knn_model.kneighbors(user_game_matrix.iloc[user_id].
        values.reshape(1, -1), n_neighbors=10)
    cf_similar_games_idx = pd.Series(distances.flatten(), index=indices.
        flatten()).sort_values().index[0:]

    # Get the titles corresponding to the recommended game indices
    recommended_titles = data.iloc[cf_similar_games_idx[:10]]['title'].tolist
        ()

    print("Games the person actually plays:")
    user_played_games = user_game_matrix.iloc[user_id][user_game_matrix.iloc[
        user_id] > 0].index.tolist()
    for i, title in enumerate(user_played_games, start=1):
        print(f"{i}. {title}")

    print("\n")
    # Print the recommendations with indices or ranks
    print("Recommended Games:")
    for i, title in enumerate(recommended_titles, start=1):
        print(f"{i}. {title}")
```

Listing 4: Recommendation Function

This function takes a user ID as input, calculates the cosine similarity between the input user and others, and uncovers the gaming gems enjoyed by similar users. The recommendations are twofold:

Games the Person Actually Plays: A showcase of the games the user has already played, providing a snapshot of their gaming journey.

Recommended Games: A curated list of game titles based on the preferences of users with similar gaming profiles.

In essence, our collaborative filtering system taps into the collective wisdom of the gaming community, guiding users towards titles enjoyed by those who share their passion.

And there you have it – the underlying mechanics of our collaborative filtering game recommendation system.

4.4 Trying to solve our memory issue

For a more comprehensible understanding we choose to display the same user as our result. Figure 5 shows our first output of our game recommender. Note that this is a result from only using 300 000 of our data rows, see *Listing 1*. When using our memory reduction function, see figure 3, we

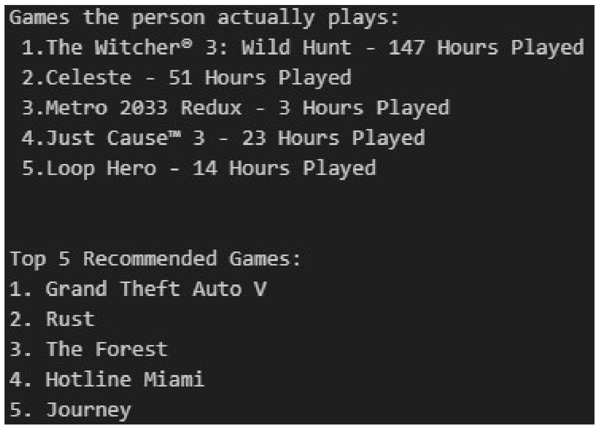

```
Games the person actually plays:
 1.The Witcher® 3: Wild Hunt - 147 Hours Played
 2.Celeste - 51 Hours Played
 3.Metro 2033 Redux - 3 Hours Played
 4.Just Cause™ 3 - 23 Hours Played
 5.Loop Hero - 14 Hours Played


Top 5 Recommended Games:
1. Grand Theft Auto V
2. Rust
3. The Forest
4. Hotline Miami
5. Journey
```

Figure 5: First Output: 300 000 rows

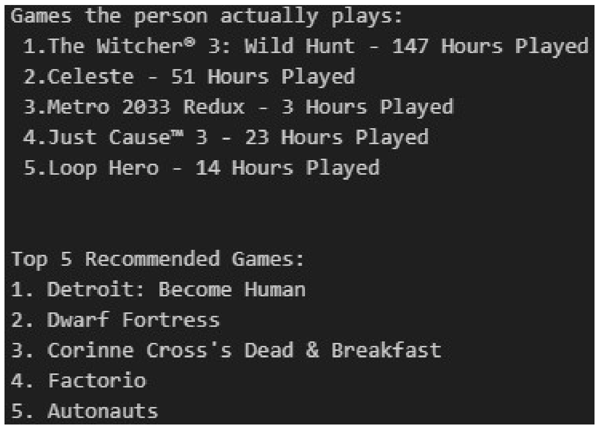
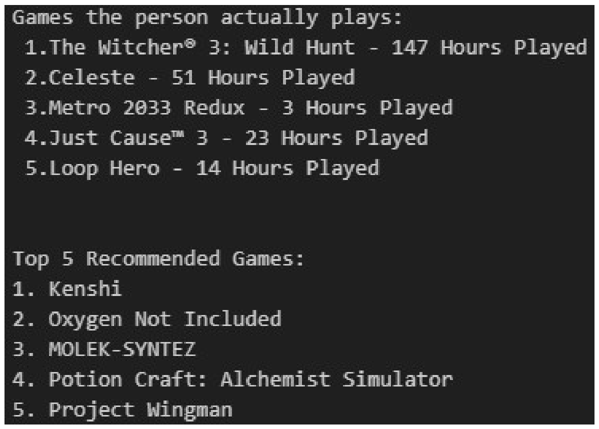

```
Games the person actually plays:
 1.The Witcher® 3: Wild Hunt - 147 Hours Played
 2.Celeste - 51 Hours Played
 3.Metro 2033 Redux - 3 Hours Played
 4.Just Cause™ 3 - 23 Hours Played
 5.Loop Hero - 14 Hours Played


Top 5 Recommended Games:
1. Detroit: Become Human
2. Dwarf Fortress
3. Corinne Cross's Dead & Breakfast
4. Factorio
5. Autonauts
```

Figure 6: First Output: 600 000 rows

where able to reach up to 900 000 rows without memory issues. Figure 6 (600 000 rows) and figure 7 (900 000 rows) shows the difference in results when increasing the number of rows. Clearly increasing the number of rows changes our result, but in this case the result seems to be completely different (which is not that surprising). However only the user itself can decide if these results are good or not, the only logical thing is that more data leads to a better game-recommender. To further increase the use of our data-set we choose to process the data in chunks, rather than the hole data-set at once. This method definitely worked better, but still the whole data-set could not be used. But with this method we where able to get up to 2.5 million rows. Listing 5 and Listing 6 shows how this was done.

```
import os
import pandas as pd

chunk_size = 500000
total_rows = len(data)
num_loops = total_rows // chunk_size + (1 if total_rows % chunk_size != 0 else
```



```
Games the person actually plays:
 1.The Witcher® 3: Wild Hunt - 147 Hours Played
 2.Celeste - 51 Hours Played
 3.Metro 2033 Redux - 3 Hours Played
 4.Just Cause™ 3 - 23 Hours Played
 5.Loop Hero - 14 Hours Played


Top 5 Recommended Games:
1. Kenshi
2. Oxygen Not Included
3. MOLEK-SYNTEZ
4. Potion Craft: Alchemist Simulator
5. Project Wingman
```

Figure 7: First Output: 900 000 rows

```python
    0)

pivot_tables = []

try:
    for i in range(num_loops):
        start_idx = i * chunk_size
        end_idx = min((i + 1) * chunk_size, total_rows)

        chunk_data = data.iloc[start_idx:end_idx]
        user_game_matrix_chunk = chunk_data.pivot_table(index='user_id',
            columns='title', values='hours', fill_value=0)

        pivot_tables.append(user_game_matrix_chunk)

        print(f"Processed chunk {i}")

except MemoryError:
    print("MemoryError occurred. Saving the processed chunks before the error.
        ")
```

Listing 5: Pivot in Chunks

```python
from scipy.sparse import vstack
aligned_chunks = []

for i, user_game_matrix_chunk in enumerate(pivot_tables):
    try:
        chunk_csr_matrix = csr_matrix(user_game_matrix_chunk.values)

        aligned_chunks.append(chunk_csr_matrix)

        print(f"Processed chunk {i}")
    except Exception as e:
        print(f"Error processing chunk {i}: {e}")
        break

if aligned_chunks:
    sparse_user_game_matrix = vstack(aligned_chunks)
    print("Completed processing all chunks")
else:
    print("No chunks were processed due to errors")
```

Listing 6: Convert to sparse matrix in chunks

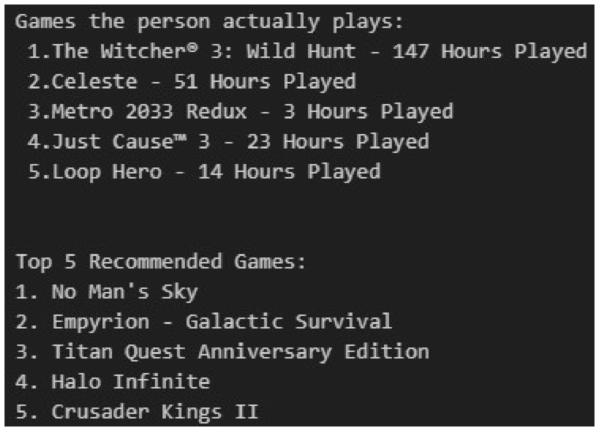
So with this method of handling our data in chunks, the main function that recommends games had to be edited a bit, see Listing 7.

```python
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.neighbors import NearestNeighbors
from scipy.sparse import vstack
from scipy.sparse import csr_matrix
from scipy.sparse import coo_matrix

max_columns = max(matrix.shape[1] for matrix in aligned_chunks)
aligned_matrices = []

for matrix in aligned_chunks:
    if matrix.shape[1] < max_columns:
        extra_columns = max_columns - matrix.shape[1]
        zero_columns = coo_matrix((matrix.shape[0], extra_columns), dtype=
            matrix.dtype)
        aligned_matrix = hstack([matrix, zero_columns])
    else:
        aligned_matrix = matrix
    aligned_matrices.append(aligned_matrix)

X_train = vstack(aligned_matrices)
knn_model = NearestNeighbors(metric='cosine', algorithm='brute')
knn_model.fit(X_train)

def game_Recommender(user_id, X_train):
    if not isinstance(X_train, csr_matrix):
        X_train = X_train.tocsr()

    distances, indices = knn_model.kneighbors(X_train[user_id].reshape(1, -1),
         n_neighbors=10)
    cf_similar_games_idx = pd.Series(distances.flatten(), index=indices.
        flatten()).sort_values().index[1:]

    recommended_games_info = data.iloc[cf_similar_games_idx[:10]][['title', '
        hours']]
    print("Games the person actually plays:")
    counter = 1
    user_played_games_info = data[data['user_id'] == user_id][['title', 'hours
        ']]
    for i, row in user_played_games_info.iterrows():
        rounded_hours = round(row['hours'])
        print(f" {counter}.{row['title']} - {rounded_hours} Hours Played")
        counter += 1
    print("\n")
    print("Top 5 Recommended Games:")
    for i, title in enumerate(recommended_games_info['title'], start=1):
        print(f"{i}. {title}")
```

Listing 7: Final Recommendation function

5 Result

Now our system has been put to the test, and the results are in. Discover how our recommendations can elevate your gaming experience. Did we



```
Games the person actually plays:
 1.The Witcher® 3: Wild Hunt - 147 Hours Played
 2.Celeste - 51 Hours Played
 3.Metro 2033 Redux - 3 Hours Played
 4.Just Cause™ 3 - 23 Hours Played
 5.Loop Hero - 14 Hours Played


Top 5 Recommended Games:
1. No Man's Sky
2. Empyrion - Galactic Survival
3. Titan Quest Anniversary Edition
4. Halo Infinite
5. Crusader Kings II
```

Figure 8: Ouput: 2.5 Million rows

nail it or unleash a hidden gem? Only one way to find out! Figure 8 shows the final result for user13.

5.1 Evaluation

To evaluate if our result is good or not is hard considering that only users in our data-set can be recommended games. The users themselves are the best evaluators in this scenario and since the users in our data-set are anonymous we had to take a different approach. Our approach was simple for evaluating our result, we asked around. First of all we choose a user that plays at least 2 games that the person we asked plays. With the result of this user we asked the following question: *How many of the recommended games do you believe were suitable for the user?*. See table 1 for the results of this evaluation.

Person	Number of suitable games
1	2/5
2	1/5
3	3/5
4	2/5
5	2/5
6	4/5

Table 1: Result from our interviews

6 Discussion

What worked, what didn't, and the lessons we've learned along the way. Join the conversation and share your thoughts. Together, let's explore the future of gaming recommendations.

First and foremost we learned that working with a big dataset does has some downsides. We found that testing our functions became dull and time consuming at times, even though we only used a chunk of our dataset. We would recommend, for anyone who is interested, to choose a smaller dataset to faster test all kinds of crazy ideas you might have. The result from this dataset could then be used on a bigger dataset. And also we would recommend to choose a small dataset nonetheless, since testing different training models could take between 2-12 hours.

Sadly the evaluation part of this project is what makes it hard to decide if this game-recommender is good or not. But even though we lack a perfect evaluation, we think the results from our own evaluation points to this algorithm being viable in some cases. The algorithm is quite simple compared to other algorithms, which makes it a perfect starting point for learning about constructing your own game-recommender. In our opinion one good recommended game is a win, and therefore we see this project as a success. In practice, the best evaluation would of course be the users themselves opinion involved in the evaluation.

7 Conclusion

In conclusion, our journey into gaming recommendations has been fruitful. Despite challenges with large datasets, our Steam Game Recommendation System stands ready to transform the gaming experience. While our evaluation methods may not be perfect, feedback suggests the algorithm's viability. Join us in revolutionizing gaming with personalized recommendations. The possibilities are endless, and the adventure awaits.

8 References

- https://medium.com/@william.alexander23326/steam-game-recommendation-system-875a4912a468, Accessed: 2023/11/29
- Database: https://www.kaggle.com/datasets/antonkozyriev/game-recommendatio on-steam, Accessed: 2023/12/01