Exploring Game Embeddings

Jonah Shader

Steam is a video game digital distribution service and storefront. It is the largest game storefront with over 50,000 games[1]. With all these games, it is impractical for a user to visit each store page individually to determine which games are worth trying. In this paper, I will explore the idea of encoding a game's description as a fixed size vector to be used in various algorithms for exploring the vast game space.

I. Introduction

Steam has a standard search engine built into their store that allows users to find games that match the search term or tag [1]. In addition, there are also means of narrowing the results by both quantitative and qualitative filters. For example, you can filter by minimum and maximum price. You can also "narrow by tag" by selecting only games with the tags: indie, action, singleplayer, adventure, etc. These tools are great for when a user can describe their desired game in these terms. However, these terms may be inconvenient for finding the desired game. One potentially useful tool would be sorting games on an arbitrary user defined axis. For example, a user could define a violence axis by selecting a handful of violent examples and a handful of non-violent examples. With embeddings, it is possible to train a model to understand the differences between these two clusters and evaluate new games as being closer to one cluster or the other. This gives users the ability to define arbitrary axes that can be used to sort games.

Another interesting idea is allowing users to perform math on these game embeddings to discover new games. This idea of performing math on embeddings has been thoroughly explored with word2vec models [2]. For example, the embedding of 'woman' minus 'man' would be similar to 'queen' minus 'king' as shown in figure 1. Perhaps this same concept could be applied to game embeddings to mix and match the embeddings of various games to find new games that have a similar embedding.

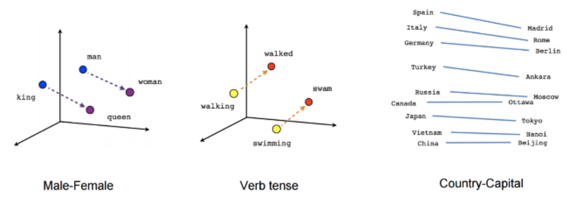

Figure 1: Examples of word2vec's embeddings having coherent representations

II. Data Collection

Given Steam's great popularity, it was not difficult to find resources on scraping Steam store data for this project, and I followed an excellent article written by Nik Davis titled "Gathering Data form the Steam Store API using Python" [3]. He explains that various services are necessary to gather the desired data. First, we need an app list that lists all the valid steam store apps. This is distinguished from the set of all non-negative integers because not every non-negative integer matches a valid steam app (since apps can become unavailable, or are non-game apps). To get this list, we use the website SteamSpy. This website provides additional data that cannot be obtained via the Steam Store API, one of which is the app id list. This list can be obtained by performing a get request to the URL https://steamspy.com/api.php with the parameters "request" and "all". Unfortunately, some undocumented change resulted in only 1000 app ids being downloaded instead of all valid games. Fortunately, for this project 1000 samples are enough since we are not training language models from scratch.

With these app ids, we can now download Steam store page data from each app id by making a get request to the URL http://store.steampowered.com/api/appdetails/ with the parameter {"appids": appid} for a particular app id. This is repeated, in batches, for every app id obtained earlier until all data is downloaded. The data is then saved to disk via "DictWriter" from the csv python package for later use [4].

III. Data Analysis

There is a lot of data provided by the Steam API that is collected, but only a subset needs to be analyzed for this project. As explained later, we will be using a Masked Language Model (MLM) to transform documents into vector representations. One important thing to consider now is the 512 token limit of this model, where a token is an English word. There are three document attributes from the data, which are short_description, about_the_game, and detailed_description. To determine which ones could be used by the MLM, histograms of character length were produced.

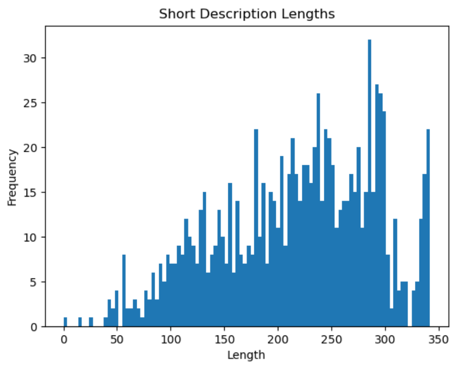

Figure 2: Histogram of character length of the "short description" attribute

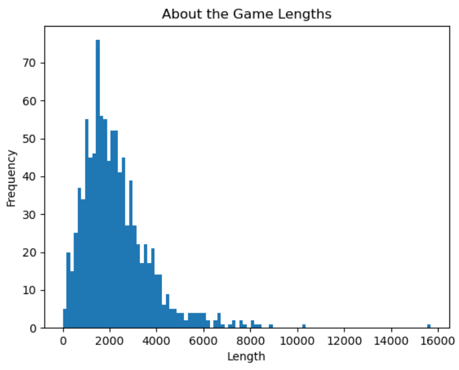

Figure 3: Histogram of character length of the "about the game" attribute

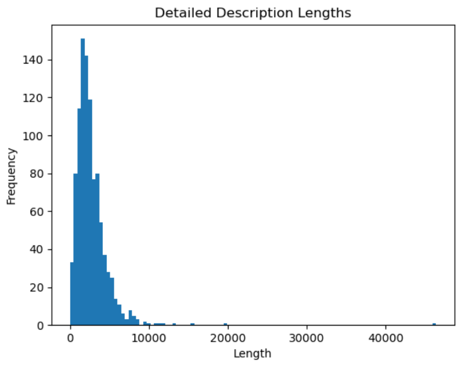

Figure 4: Histogram of character length of the "detailed description" attribute

From figures 2, 3 and 4, we see that the short description attribute will easily fit within our MLM model as the largest value of 350 character is much smaller than 512 tokens. If we consider the average token to account for roughly 6.5 characters, we find that our MLM can only handle roughly 512 * 6.5 = 3328 characters. Anything greater than this must be truncated, and the resulting embedding may suffer. Most documents from "about the game" lies below 3328 characters, so its use may be considered alongside the "short description" attribute.

Matrix Showing Proportion of Duplicates			
	Short Description	About the Game	Detailed Description
Short Description	1	0.018	0.018
About the Game	0.018	1	0.623
Detailed Description	0.018	0.623	1

Since a large proportion of "about the game" overlaps with "detailed description", the latter was decided to be less favorable due to the high duplicate rate and larger outliers.

Besides the documents, other attributes were used to produce a "custom" game embedding. These attributes include required_age, price, categories, and genres. Required_age is a scalar, and missing values can be interpreted as zero (no age requirement). Price has many fields or "sub-attributes" such as currency, initial, and final. Since the vast majority of games use USD as the currency, non-USD currencies were replaced with the average USD price instead of converting them. Only the final price is used, with initial price being unused. The last two attribute also contain sub-attributes; however, these ones are not of fixed length. They are lists of dictionaries of id and description. For example, a single game's genre attribute may contain [{'id': 1, 'description': 'Action'}], which is a list of length 1. Games can contain multiple genres or categories, so we first found every possible category and genre and used that to determine the final vector length. Then, as we iterate through the data, we mark a category or genre as 1 if it is found or 0 if it is not. Any remaining nans are replaced by the mean, followed by normalization.

IV. Data Preprocessing

Applying the MLM to the document attributes is considered pre-processing because it occurs prior to the training of models. The MLM can be treated as a black box since there will not be any gradients passing through it or changes made to the model. The main benefit of applying the MLM ahead of time is saving computation.

Upon further inspection of the document attributes, many html tags can be found. Looking at a steam page, we see that game descriptions contain lots of additional formatting like bullet points, images, titles, and links. It is unclear how these are interpreted by the MLM tokenizer, so variation of game embeddings were produced with and without the removal of html components. In the end we have four variations: (about_the_game, short_description) x (html removed, raw).

To compare the quality of these game embeddings, we trained linear regression models that map the game embeddings to the custom embedding that was produced earlier. Their mean squared error are displayed in the following table.

Configuration	Mean Squared Error
Short Description, Raw	0.1454
Short Description, HTML Cleaned	0.1456
About the Game, Raw	0.2570
About the Game, HTML Cleaned	0.1577
Above 4 concatenated	3.074e-7

We see that cleaning the HTML has minimal effect on the Short Description embeddings, whereas it has a substantial effect on the About the Game embeddings. As a quick experiment, all four embeddings were combined to produce one large game embedding matrix, and this somehow produced a mean squared error six orders of magnitude smaller than the next smallest error. One possible explanation of this is that with the concatenated version, there are more attributes than samples, which may cause overfitting to occur. I was unaware that linear models could overfit like this. More experiments need to be done to determine the cause of this.

After producing these game embeddings, we can produce additional visualizations to inspect them. We produce plots of PCA and t-SNE in two dimensions. For colorization, we can make use of attributes that were used to make the custom embeddings. Since these vectors are quite large (80+), we will only use a subset for colorization. If a game has 'Action' as one of its genres, it will be colored red. If not, then it will be colored blue.

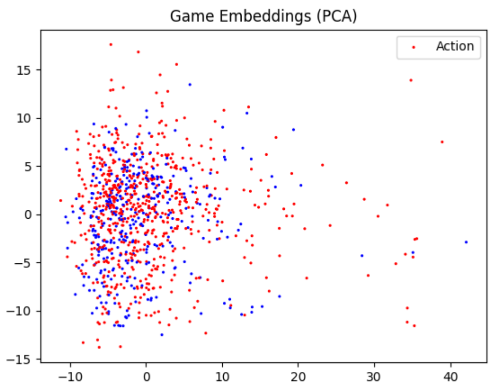

Figure 5: 2D PCA plot of 4 concatenated game embeddings

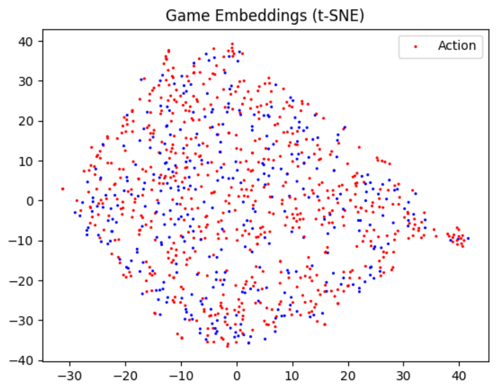

Figure 6: 2D t-SNE plot of 4 concatenated game embeddings

As we can see, there does not appear to be any clustering for the 'Action' genre. At this point, it is unclear if this indicates poor embedding quality or if these embeddings simply do not represent the 'Action' genre strongly enough for visible clusters to form.

V. Models, Methods, Experiments

Earlier it was discussed that a Masked Language Model (MLM) was to be used for translating the game's descriptions from the Steam Store to fixed length vectors. The model architecture that is used is called Bidirectional Encoder Representations from Transformers (BERT) [5]. We are using HuggingFace's "bert-base-uncased" pre-trained model, which does not distinguish between Cased and uncased text [6]. The model outputs various tensors, but the one we are interested in is the text features which is a vector of length 768, representing the input document. Applying this to the four configurations across our 1000 game samples, we get a matrix of size 1000x(768*4).

The first experiment was to try identifying similar games by sorting by Euclidian distance between game vectors. To do this, we first lookup the game embedding for our desired game, and then compute the distance between it and all other game embeddings. Let us use the video game Factorio as an example. From the Steam page: "*Factorio is a game about building and creating automated factories to produce items of increasing complexity, within an infinite 2D world. Use your imagination to design your factory, combine simple elements into ingenious structures, and finally protect it from the creatures who don't really like you.* " [7]. From this description, we would expect other similar games like Satisfactory[8] or Dyson Sphere Program[9], which are both factory building games, to have embeddings that are closer in Euclidian space compared to other different games. However, when we sort each game embedding by Euclidian distance, we get the following top 10 similar games:

Top 10 Games Similar to Factorio		
Rank	Name	Euclidian Distance
1	BioShock	7.21
2	BattleBlock Theater	7.82
3	Layers of Fear (2016)	7.85
4	Eternal Return	7.92
5	RUNNING WITH RIFLES	7.93
6	PAYDAY 2	7.97
7	DARK SOULS™: Prepare To Die™ Edition	8.06
8	ArcheBlade™	8.15
9	Risk of Rain	8.31

Upon further investigation, none of these games are about building factories, and are very different from Factorio in other aspects.

The next experiment was to try performing math on these game embeddings to discover new games. The example tried is "Factorio + Risk of Rain 2 – Risk of Rain = Satisfactory". The logic is Factorio is a 2d factory game, Risk of Rain 2 is a 3D version of Risk of Rain 1, so Risk of Rain 2 minus Risk of Rain 1 could represent 2D to 3D. Adding this to Factorio in theory would yield a game embedding representing a 3D factory building game, which is exactly what Satisfactory is. However, the results were equally disappointing as the previous experiment, so the resulting table of similar games will not be shown.

The final experiment was to define an axis between two user defined categories for sorting or classifying games. This was done by selecting a group of games with characteristics A, and selecting another group of games with characteristics B. We assign each game embedding in group A the target output of -1 and each embedding in group B the target output of 1 and then perform linear regression. This model can then be used to either sort games on this axis that intercepts A and B, or it can be used to classify games as belonging to A or B. For an example, we made an axis that should distinguish factory games from non-factory games. For group A, we selected Factorio and Satisfactory (both of which are factory building games). For group B, we selected Risk of Rain 1 and Team Fortress 2 (both of which are non-factory building games). Using this model to predict the label of Dyson Sphere, we get a value of -1.005, which is good because Dyson Sphere is a building game and is similar to Factorio and Satisfactory. The prediction for Call of Duty, a non-factory building game, evaluates to 0.487, which is good because Call of Duty is closer to the non-building games. However, the predicted value of Risk of Rain 2 is -0.176, which is bad because Risk of Rain 2 is not a factory building game. Since this is a user-defined axis, we would have to manually label each game as factory building or non-factory building to properly evaluate this technique. Fortunately, we can make use of the custom embeddings to make groups from an existing axis. We will select two groups based on if their genre contains 'Action' or not. We gather all action and non-action games and truncate the larger of the two so that each group has 345 games. We take

80% for training and 20% for testing. The results show that the training accuracy is 1.0 and the testing accuracy is 0.692. The training accuracy of 1.0 is a clear indication of overfitting, and the results will be interpreted further in the conclusion.

VI. Conclusion

Of the three experiments, only one showed any meaningful result. In the last experiment, we found 1.0 training accuracy while only getting 0.692 test accuracy. This is a clear indication of overfitting, and it is likely due to the size of the concatenated game embeddings being much larger than the number of samples. Similarly, the linear model mapping this concatenated game embeddings to the custom embeddings yielded an MSE of 3.074e-7, which is much smaller than expected. Perhaps this overfitting problem could be mitigated by first performing PCA on the embeddings to reduce them to a more reasonable size so that there are more samples than attributes. In a final product, this could be performed on the fly depending on the size of the user's groups for the custom axis.

Perhaps one could take the average of a game's user review embeddings and use that as the game embedding. This could be beneficial because game descriptions can be written in various styles, whereas user reviews are written by multiple people, meaning the average embedding would cancel out any encoding of written style.

Image Sources:

Figure 1: https://thegradient.pub/nlp-imagenet/

Source Code: https://github.com/jonahshader/steam_game_embedding

References:

[1] https://store.steampowered.com/search/

[2] https://en.wikipedia.org/wiki/Word2vec

[3] https://nik-davis.github.io/posts/2019/steam-data-collection/

[4] https://docs.python.org/3/library/csv.html

[5] https://arxiv.org/abs/1810.04805

[6] https://huggingface.co/bert-base-uncased

[7] https://store.steampowered.com/app/427520/Factorio/

[8] https://store.steampowered.com/app/526870/Satisfactory/

[9] https://store.steampowered.com/app/1366540/Dyson_Sphere_Program/