2025 Machine Learning Project

The Steam Success Predictor: A Multi-Model Approach

Hongxin Li | Yuenyuen Yu | Difu Chen | Roshi Bhati | Krish jani
hl3714 yy6012 dc5902 rb6161 kj2743

Project Overview

This project investigates the drivers of commercial success on the Steam platform using a dataset updated in 2025 on Kaggle. We designed our labs through 4 progressive modeling phases, from traditional machine learning methods (Random Forest and XGBoost) to K-class and finally to advanced Deep Learning architectures. Our goal was to predict both the relative popularity tier of a game and its specific peak concurrent user count. Ultimately, our optimized Neural Network achieved a classification accuracy of over **60.5%**, and our regression model successfully predicted player activity trends with a mean absolute error of **0.44** on a log scale.

Insights From Our Analysis

1. Pre-release metadata, such as language support and pricing, serves as a reliable proxy for production quality, allowing us to filter out low-effort titles even without gameplay data.
2. Deep Learning models proved superior to linear baselines because they can effectively capture complex combinations of genre tags and features that simpler models miss.
3. We identified a Middle Class Trap: our models could easily identify high popularity and low popularity, however, the "average" games proved the most difficult to predict, for they lack the distinct statistical signatures found in outlier successes.

Problem Statement

The video game industry is characterized by high risk and extreme saturation. With thousands of new titles launching on Steam every month, developers and investors face a critical challenge: distinguishing potential hits from failures before significant resources are lost. In conclusion, on the business side, the data analysis of the success of a game can be critical. The objective of this project is to apply machine learning techniques to predict a game's market performance. We seek to uncover the hidden data patterns that separate successful titles from the rest of the market.

Comparative Analysis of 4 Models

Models	**Strengths**	**Weaknesses**	**Key Finding**
Pre-Realse (Random Forest)	Simulates real-world business scenarios with no data leakage	**Accuracy (51.8%):** Constrained by invisible factors like "luck" and "marketing," hitting a performance ceiling at 52%	Pricing and localization scope act as effective "quality filters" to screen out shovelware, though they are insufficient for predicting

			breakout hits
Linear Classification	Algorithm built from scratch, ensuring full mathematical transparency	**Accuracy(51.9%):** Even with strong post-release features, linear boundaries failed to separate non-linear data distributions, especially for the "Medium" class.	Simple linear models are unable to capture complex non-linear relationships between game features
Neural Network	**High Performance (60.5%):** Hidden layers successfully captured nuanced interactions between tag combinations and genres	Required significantly more computational resources	The non-linear fitting capability of neural networks was the key to raising the prediction ceiling, proving that game success follows deep statistical patterns
CCU Regression	Provides specific numerical estimates rather than vague popularity tiers	Unable to predict exact user counts (average error approx. 10,000 users)	While exact numbers are difficult to predict, metadata effectively reflects game vitality at an order-of-magnitude level

Short Conclusion

Difficulties

Our biggest challenge was the opacity of external market forces. In the current social media era, a game's success is heavily driven by viral marketing, influencer coverage, and meme culture, which are factors that are invisible to our dataset. We found that even objectively "mediocre" games can become massive hits due to sudden trends or streamer attention, creating stochastic outliers that no feature-based model can predict. Furthermore, we were constrained by the limitations of public data. To significantly improve accuracy, we would need access to private, pre-release signals such as Steam Wishlist counts, marketing budgets, and Discord community activity, which remain the "hidden variables" of game success.

Successes

We successfully demonstrated that feature engineering had a greater impact on performance than model selection alone. By advancing from linear models to Deep Learning, we broke through the accuracy ceiling, achieving a **60.5% classification accuracy** and a reliable regression error rate. This confirms that our engineered features effectively capture the latent quality of a game.

Key Takeaway

Game success on Steam is not purely random. While luck plays a role, our project proves that **intrinsic metadata serves** as a strong predictor of market performance. Deep Learning is essential for unlocking these patterns, transforming raw metadata into actionable business intelligence.

Baseline Model: K-class Logistic Regression

Target

We want to do 3-class classification for game popularity. We use estimated_owners to build the label pop_class. We rank all games by estimated_owners and split by percentile: bottom 25% is low (0), middle 50% is medium (1), top 25% is high (2).

Features (X)

After creating pop_class, we do NOT use estimated_owners as input features (to avoid label leakage). We use numeric columns like required_age, price, dlc_count, achievements, recommendations, user_score, positive, negative, peak_ccu, num_reviews_total. We also add multi-hot features from categories, so we do not create too many sparse columns.

Model

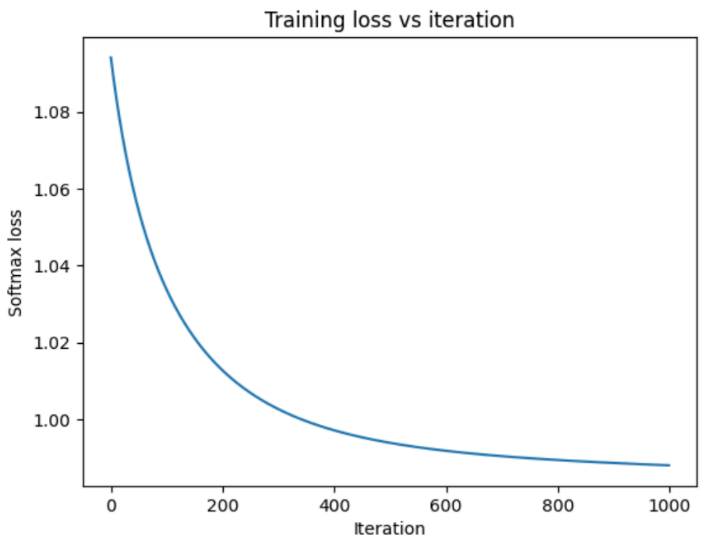
We use softmax regression (multi-class logistic regression). We train the weight matrix W with gradient descent to minimize softmax cross-entropy loss. We standardize features and add a bias term.

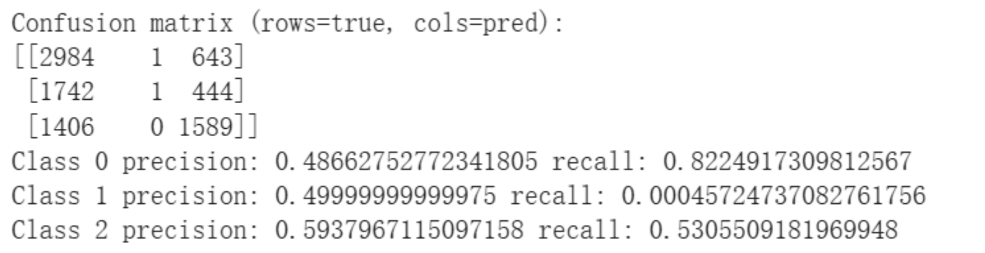
Train/Test setup

We start from 72563 games. After dropping missing values, we use 44,048 rows. We split data into 80% train and 20% test with stratify. Final input dimension is 53 features (plus 1 bias).

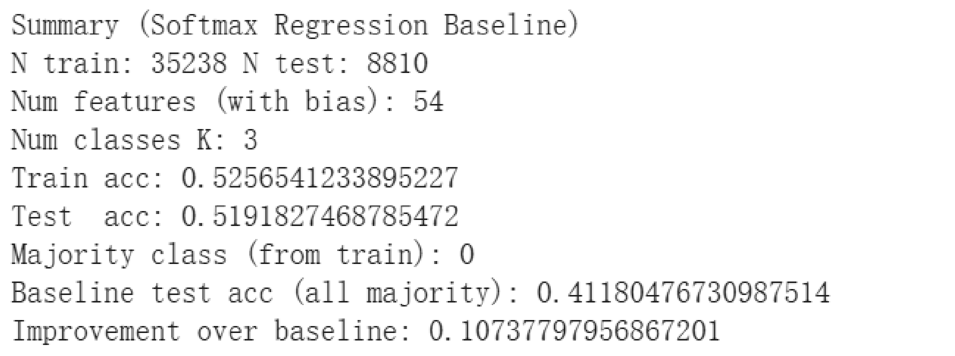
Results

Train accuracy is 0.5257 and test accuracy is 0.5192. As a simple baseline, we predict all test samples as the majority class from the training set, and baseline test accuracy is 0.4118. Our model improves baseline by about +0.1074. We also report confusion matrix and per-class precision/recall. Macro-F1 is 0.3909.





```
Confusion matrix (rows=true, cols=pred):
[[2984    1  643]
 [1742    1  444]
 [1406    0 1589]]
Class 0 precision: 0.48662752772341805 recall: 0.8224917309812567
Class 1 precision: 0.49999999999975 recall: 0.00045724737082761756
Class 2 precision: 0.5937967115097158 recall: 0.5305509181969948
```



```
Summary (Softmax Regression Baseline)
N train: 35238 N test: 8810
Num features (with bias): 54
Num classes K: 3
Train acc: 0.5256541233895227
Test  acc: 0.5191827468785472
Majority class (from train): 0
Baseline test acc (all majority): 0.41180476730987514
Improvement over baseline: 0.10737797956867201
```

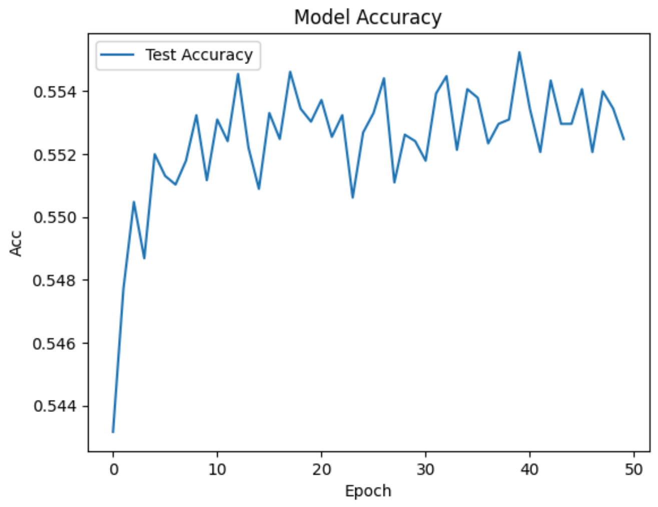
Neural Network Design

To better predict our three outputs, we built a neural network hoping for better results. In this design we now incorporated categorical data into our input through multiple-hot coding. (Adding 460 extra predictors)

After completing the build, to find the best setup for our neural network, we locked a seed for our model so every run with the same setup produces identical results.

At first we want to compare how much better the neural network does compared to the k-class regression model. (We start our neural network setup with only numerical features and a ReLu activation)

```
Epoch 5/50 | Acc: 0.5520 | Train Loss: 0.9688
Epoch 10/50 | Acc: 0.5512 | Train Loss: 0.9652
Epoch 15/50 | Acc: 0.5509 | Train Loss: 0.9641
Epoch 20/50 | Acc: 0.5530 | Train Loss: 0.9633
Epoch 25/50 | Acc: 0.5527 | Train Loss: 0.9628
Epoch 30/50 | Acc: 0.5524 | Train Loss: 0.9621
Epoch 35/50 | Acc: 0.5541 | Train Loss: 0.9619
Epoch 40/50 | Acc: 0.5552 | Train Loss: 0.9613
Epoch 45/50 | Acc: 0.5530 | Train Loss: 0.9616
Epoch 50/50 | Acc: 0.5525 | Train Loss: 0.9610
```

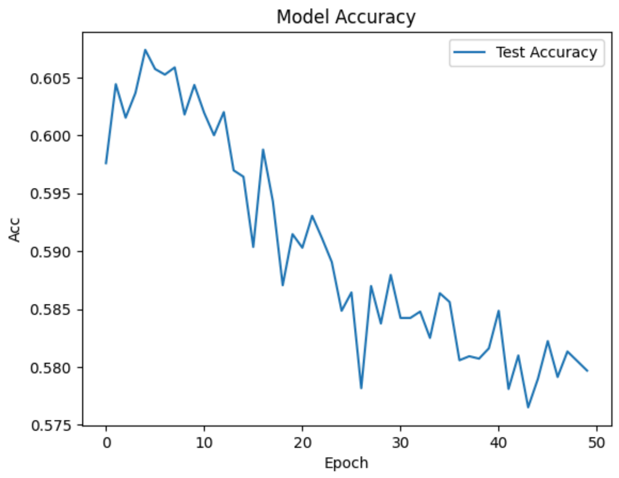


```
num_features = ['price', 'dlc_count', 'achievements', 'required_age']
cat_cols = []
```

Now we want to compare how much adding categorical data through multiple-hot coding could help our model. (Categorical data included)

```
num_features = ['price', 'dlc_count', 'achievements', 'required_age']
cat_cols = ['genres','categories','supported_languages']
```

```
Epoch 5/50 | Acc: 0.6074 | Train Loss: 0.8691
Epoch 10/50 | Acc: 0.6044 | Train Loss: 0.8404
Epoch 15/50 | Acc: 0.5964 | Train Loss: 0.8154
Epoch 20/50 | Acc: 0.5915 | Train Loss: 0.7953
Epoch 25/50 | Acc: 0.5849 | Train Loss: 0.7752
Epoch 30/50 | Acc: 0.5880 | Train Loss: 0.7592
Epoch 35/50 | Acc: 0.5864 | Train Loss: 0.7455
Epoch 40/50 | Acc: 0.5816 | Train Loss: 0.7351
Epoch 45/50 | Acc: 0.5790 | Train Loss: 0.7254
Epoch 50/50 | Acc: 0.5797 | Train Loss: 0.7156
```

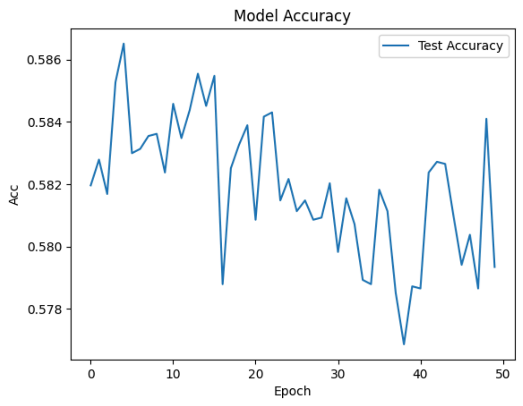


Apparently we do see better results in terms of final accuracy with more categorical parameters added. However, we also see that through more training, we only lead to worse accuracy. Here we suspected that due to the scarce and high-volume categorical data, we may be overfitting the model. Thus, we tried to only take less categorical parameters, by only using ‘categories’, ‘genres’, or ‘supported languages’. Yet, the negative or unstable training slope remained.

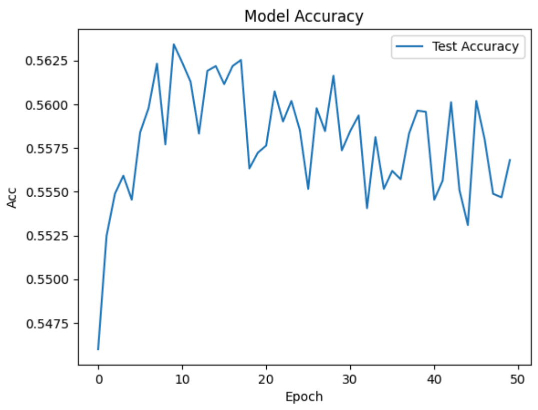
```
num_features = ['price', 'dlc_count', 'achievements', 'required_age']
cat_cols = ['supported_languages']
```

```
num_features = ['price', 'dlc_count', 'achievements', 'required_age']
cat_cols = ['categories']
```

```
Epoch 5/50 | Acc: 0.5865 | Train Loss: 0.8639
Epoch 10/50 | Acc: 0.5824 | Train Loss: 0.8620
Epoch 15/50 | Acc: 0.5845 | Train Loss: 0.8593
Epoch 20/50 | Acc: 0.5839 | Train Loss: 0.8580
Epoch 25/50 | Acc: 0.5822 | Train Loss: 0.8565
Epoch 30/50 | Acc: 0.5820 | Train Loss: 0.8551
Epoch 35/50 | Acc: 0.5788 | Train Loss: 0.8535
Epoch 40/50 | Acc: 0.5787 | Train Loss: 0.8520
Epoch 45/50 | Acc: 0.5810 | Train Loss: 0.8505
Epoch 50/50 | Acc: 0.5793 | Train Loss: 0.8488
```

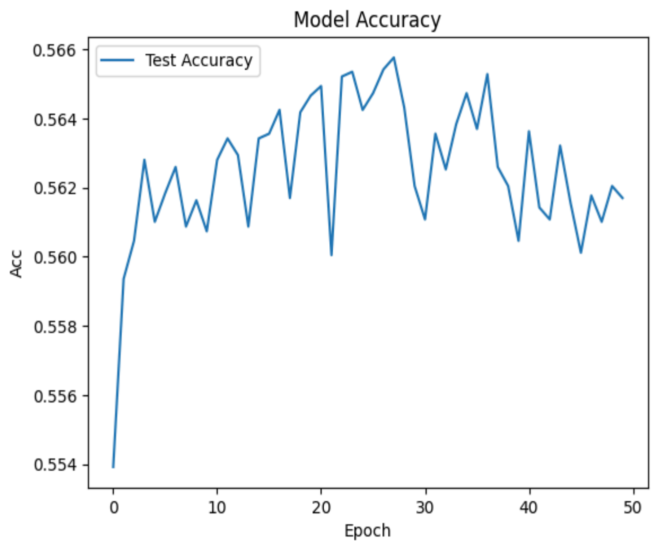


```
Epoch 5/50 | Acc: 0.5545 | Train Loss: 0.9444
Epoch 10/50 | Acc: 0.5634 | Train Loss: 0.9308
Epoch 15/50 | Acc: 0.5622 | Train Loss: 0.9223
Epoch 20/50 | Acc: 0.5572 | Train Loss: 0.9140
Epoch 25/50 | Acc: 0.5585 | Train Loss: 0.9082
Epoch 30/50 | Acc: 0.5574 | Train Loss: 0.9027
Epoch 35/50 | Acc: 0.5552 | Train Loss: 0.8987
Epoch 40/50 | Acc: 0.5596 | Train Loss: 0.8942
Epoch 45/50 | Acc: 0.5531 | Train Loss: 0.8895
Epoch 50/50 | Acc: 0.5568 | Train Loss: 0.8855
```



```
num_features = ['price', 'dlc_count', 'achievements', 'required_age']
cat_cols = ['genres']
```

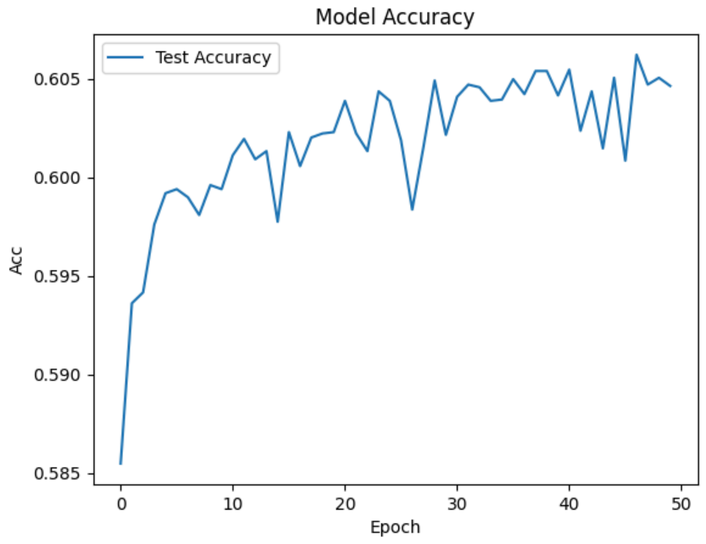
```
Epoch 5/50 | Acc: 0.5610 | Train Loss: 0.9431
Epoch 10/50 | Acc: 0.5607 | Train Loss: 0.9350
Epoch 15/50 | Acc: 0.5634 | Train Loss: 0.9298
Epoch 20/50 | Acc: 0.5647 | Train Loss: 0.9255
Epoch 25/50 | Acc: 0.5643 | Train Loss: 0.9215
Epoch 30/50 | Acc: 0.5620 | Train Loss: 0.9184
Epoch 35/50 | Acc: 0.5647 | Train Loss: 0.9156
Epoch 40/50 | Acc: 0.5605 | Train Loss: 0.9128
Epoch 45/50 | Acc: 0.5616 | Train Loss: 0.9110
Epoch 50/50 | Acc: 0.5617 | Train Loss: 0.9092
```



After failing, we realized that the ReLu activation that neglected negative weights might have been the problem, as it may have restrained unpopular categories to not have a negative influence on the game's success in which it should. Thus after switching to sigmoid we have better results, and the negative slope trend seems to be gone.

Sigmoid Activation Results

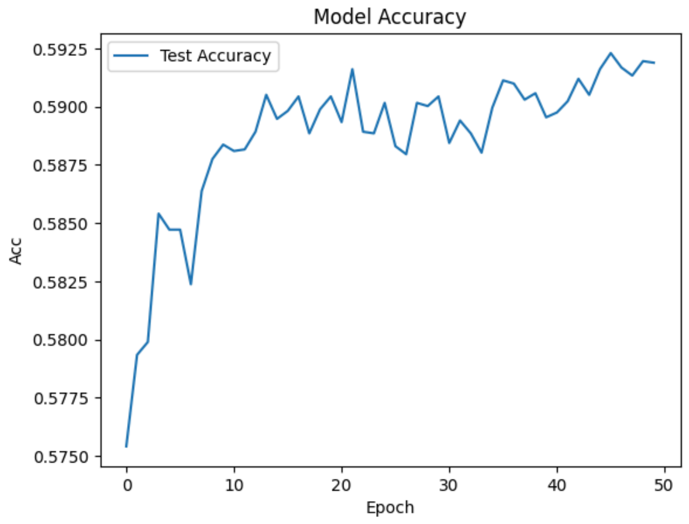
```
Epoch 5/50 | Acc: 0.5992 | Train Loss: 0.9001
Epoch 10/50 | Acc: 0.5994 | Train Loss: 0.8923
Epoch 15/50 | Acc: 0.5977 | Train Loss: 0.8860
Epoch 20/50 | Acc: 0.6023 | Train Loss: 0.8809
Epoch 25/50 | Acc: 0.6039 | Train Loss: 0.8749
Epoch 30/50 | Acc: 0.6021 | Train Loss: 0.8695
Epoch 35/50 | Acc: 0.6039 | Train Loss: 0.8644
Epoch 40/50 | Acc: 0.6041 | Train Loss: 0.8591
Epoch 45/50 | Acc: 0.6050 | Train Loss: 0.8535
Epoch 50/50 | Acc: 0.6046 | Train Loss: 0.8471
```



```
num_features = ['price', 'dlc_count', 'achievements', 'required_age']
cat_cols = ['categories','genres','supported_languages']
```

```
num_features = ['price', 'dlc_count', 'achievements', 'required_age']
cat_cols = ['categories']
```

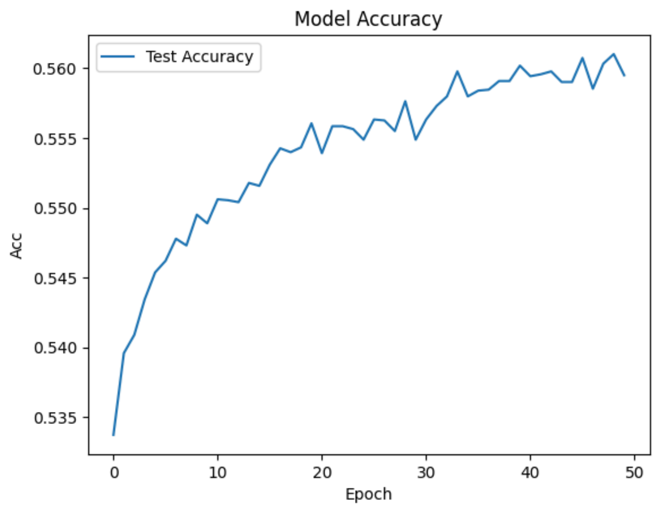
```
Epoch 5/50 | Acc: 0.5847 | Train Loss: 0.9294
Epoch 10/50 | Acc: 0.5884 | Train Loss: 0.9232
Epoch 15/50 | Acc: 0.5895 | Train Loss: 0.9197
Epoch 20/50 | Acc: 0.5904 | Train Loss: 0.9174
Epoch 25/50 | Acc: 0.5902 | Train Loss: 0.9155
Epoch 30/50 | Acc: 0.5904 | Train Loss: 0.9140
Epoch 35/50 | Acc: 0.5900 | Train Loss: 0.9121
Epoch 40/50 | Acc: 0.5895 | Train Loss: 0.9110
Epoch 45/50 | Acc: 0.5916 | Train Loss: 0.9096
Epoch 50/50 | Acc: 0.5919 | Train Loss: 0.9086
```



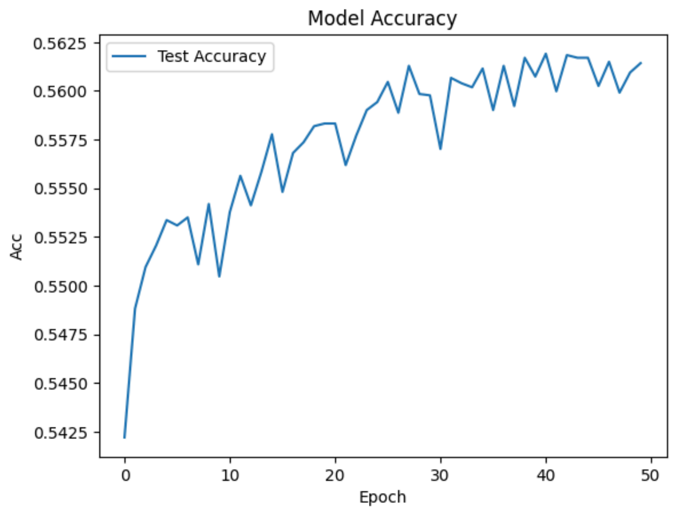
```
num_features = ['price', 'dlc_count', 'achievements', 'required_age']
cat_cols = ['supported_languages']
```

```
num_features = ['price', 'dlc_count', 'achievements', 'required_age']
cat_cols = ['genres']
```

```
Epoch 5/50 | Acc: 0.5454 | Train Loss: 0.9775
Epoch 10/50 | Acc: 0.5489 | Train Loss: 0.9676
Epoch 15/50 | Acc: 0.5516 | Train Loss: 0.9613
Epoch 20/50 | Acc: 0.5561 | Train Loss: 0.9555
Epoch 25/50 | Acc: 0.5549 | Train Loss: 0.9507
Epoch 30/50 | Acc: 0.5549 | Train Loss: 0.9464
Epoch 35/50 | Acc: 0.5580 | Train Loss: 0.9433
Epoch 40/50 | Acc: 0.5602 | Train Loss: 0.9406
Epoch 45/50 | Acc: 0.5590 | Train Loss: 0.9381
Epoch 50/50 | Acc: 0.5595 | Train Loss: 0.9359
```



```
Epoch 5/50 | Acc: 0.5534 | Train Loss: 0.9658
Epoch 10/50 | Acc: 0.5505 | Train Loss: 0.9589
Epoch 15/50 | Acc: 0.5578 | Train Loss: 0.9554
Epoch 20/50 | Acc: 0.5583 | Train Loss: 0.9523
Epoch 25/50 | Acc: 0.5594 | Train Loss: 0.9498
Epoch 30/50 | Acc: 0.5598 | Train Loss: 0.9481
Epoch 35/50 | Acc: 0.5612 | Train Loss: 0.9463
Epoch 40/50 | Acc: 0.5607 | Train Loss: 0.9449
Epoch 45/50 | Acc: 0.5617 | Train Loss: 0.9438
Epoch 50/50 | Acc: 0.5614 | Train Loss: 0.9425
```

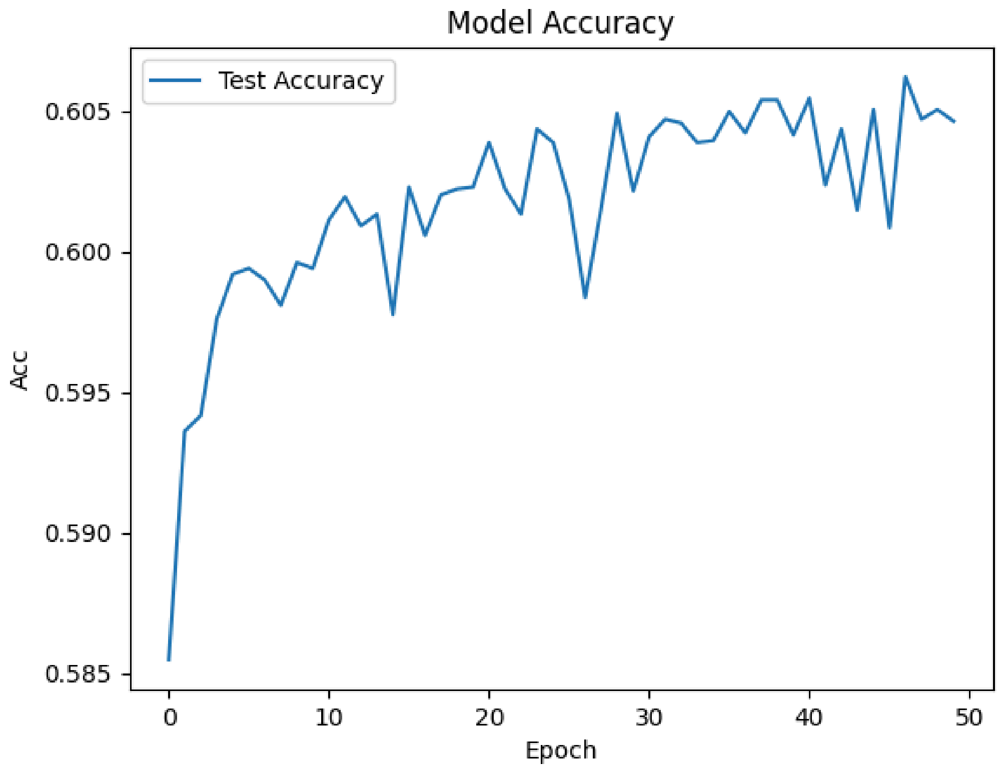


Final Model

Our final model uses a 2 layer neural network that performs sigmoid activations, and uses the numerical parameters, price, dlc_count, achievements, required_age, as well as categorical parameters, categories, genres, and supported_languages. We managed to get an accuracy of 0.6046 for a 3-class classification of user number prediction based on pre-game release statistics. (We did not use parameters such as number of reviews, that would be only available after the game is released, which would be impossible for users to have)

```
num_features = ['price', 'dlc_count', 'achievements', 'required_age']
cat_cols = ['categories','genres','supported_languages']
```

```
Epoch 5/50 | Acc: 0.5992 | Train Loss: 0.9001
Epoch 10/50 | Acc: 0.5994 | Train Loss: 0.8923
Epoch 15/50 | Acc: 0.5977 | Train Loss: 0.8860
Epoch 20/50 | Acc: 0.6023 | Train Loss: 0.8809
Epoch 25/50 | Acc: 0.6039 | Train Loss: 0.8749
Epoch 30/50 | Acc: 0.6021 | Train Loss: 0.8695
Epoch 35/50 | Acc: 0.6039 | Train Loss: 0.8644
Epoch 40/50 | Acc: 0.6041 | Train Loss: 0.8591
Epoch 45/50 | Acc: 0.6050 | Train Loss: 0.8535
Epoch 50/50 | Acc: 0.6046 | Train Loss: 0.8471
```



Final Conclusion: Data Driven Insights

Based on the feature importance analysis from our Random Forest model and the weight adjustments in our Neural Network, we can construct a vague profile for what constitutes a high-potential game versus a low-performing one.

The High popularity Profile:

- High Investment Signals: The strongest pre-release predictor is "effort." High popularity games typically support a high number of languages (often >10) and multiple platforms.
- Community Stickiness: In our deep learning models, the ratio of reviews to owners was a critical indicator. A hit game triggers a high desire to communicate.Players don't just buy it, they talk about it.
- Complex Tagging: Successful titles rarely rely on generic tags like "Action" or "Indie" alone. They feature complex, multi-dimensional tag combinations, indicating rich gameplay mechanics that target specific, engaged niches.

The Low Popularity Profile:

- Minimum Viable Metadata: Games predicted as low popularity often feature minimal localization and generic categorization
- Lack of Post-Launch Support: A strong correlation was found between low user counts and zero DLC content or lack of Steam community features, signaling a "release and abandon" strategy by developers

Future Outlook

While our Neural Network achieved a commendable **60.5%** accuracy using only metadata, we recognize that we have reached the limit of what public data can predict. To bridge the gap to 80%+ accuracy, future iterations of this project would require access to the Hidden Variables of the industry:

- Pre-Release Hype Metrics: Unlocking Steam Wishlist counts and Follower numbers would allow the model to quantify anticipation, which is often the biggest driver of first day sales.
- Marketing: Integrating data from social media, Twitch viewership trends, and estimated marketing budgets would help the model account for the viral factor that metadata ignores.