The Steam Success Predictor: A Multi-Model Approach

2025 Machine Learning Project

Team Members:

- Hongxin Li (hl3714)
- Yuenyuen Yu (yy6012)
- Difu Chen (dc5902)
- Roshi Bhati (rb6161)
- Krish Jani (kj2743)

Table of Contents

---

1. Project Overview & 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.

The Hypothesis: We believe that a game's market potential is imprinted in its pre-release metadata.

Our Approach: We designed a progressive modeling pipeline to test this hypothesis. We moved from simple linear baselines to complex deep learning architectures to understand *how* different data structures (pricing, tags, language support) interact to create a "hit."

2. Phase 1: Baseline (Logistic Regression)

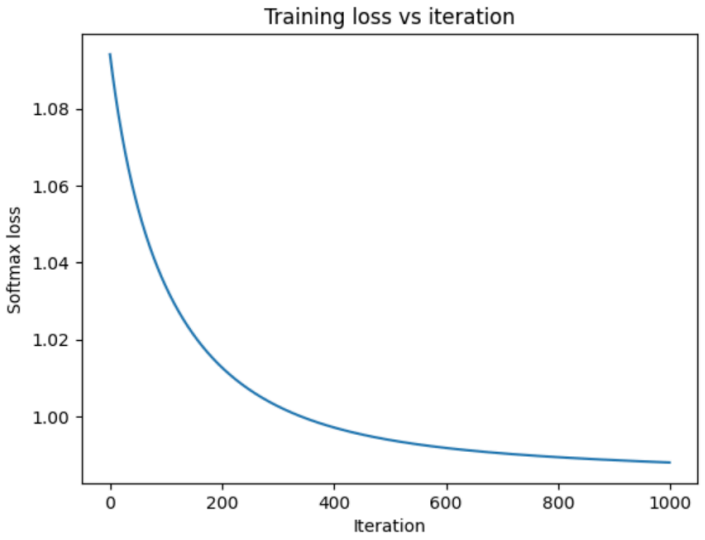
Why we chose this model:

We started with Softmax Regression (Multi-class Logistic Regression) to establish a mathematical baseline. We needed to know if game success follows a linear trend. For example, does adding one more language or increasing the price by $5 directly correlate to a proportional increase in success probability? This model offers full transparency. We can see exactly how much weight is assigned to each feature.

Model Setup:

- **Target:** 3-Class Popularity (Low, Medium, High).
- **Constraint:** Excluded `estimated_owners` to prevent data leakage.
- **Features:** 53 features, including multi-hot encoded categories.

Results:




```
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: 0.0004574737082761756
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
```

Inference & Findings:

- **The Linearity Failure:** While the model achieved ~51.9% accuracy, it failed significantly on the "Medium" popularity class.
- **Inference:** We learned that game success is **non-linear**. You cannot simply "add" features to guarantee a hit. The relationship between features is multiplicative, not additive (e.g., a high price is only good *if* the game also has high graphics tags). Linear boundaries could not capture these dependencies.

---

3. Phase 2: Pre-Release Prediction (Random Forest)

Why we chose this model:

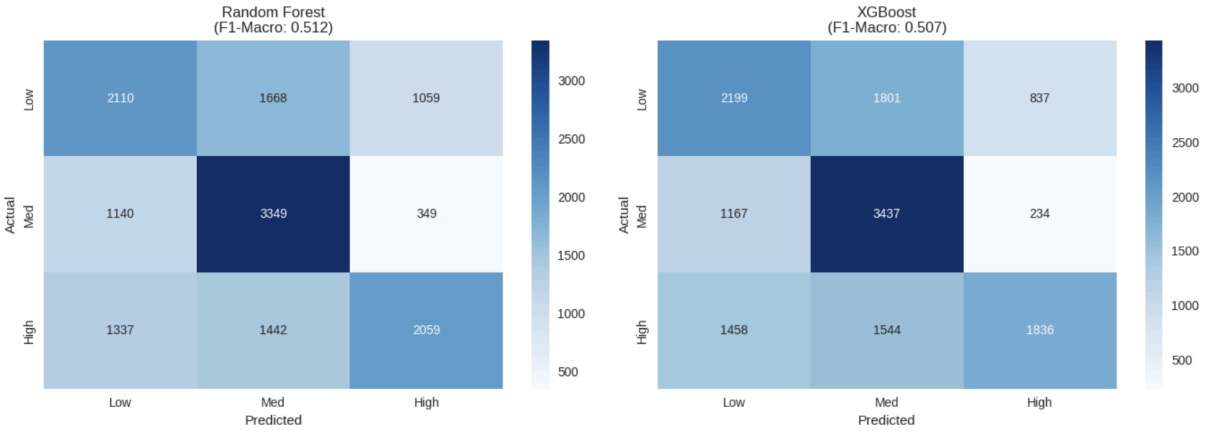
To simulate a real-world business scenario, we enforced a strict "Pre-Release" constraint. We removed all features that wouldn't exist before launch (reviews, ratings). We chose Random Forest because tree-based models excel at handling the non-linear interactions we missed in Phase 1 (e.g., distinguishing that "Indie" + "Low Price" is acceptable, but "AAA" + "Low Price" is suspicious).

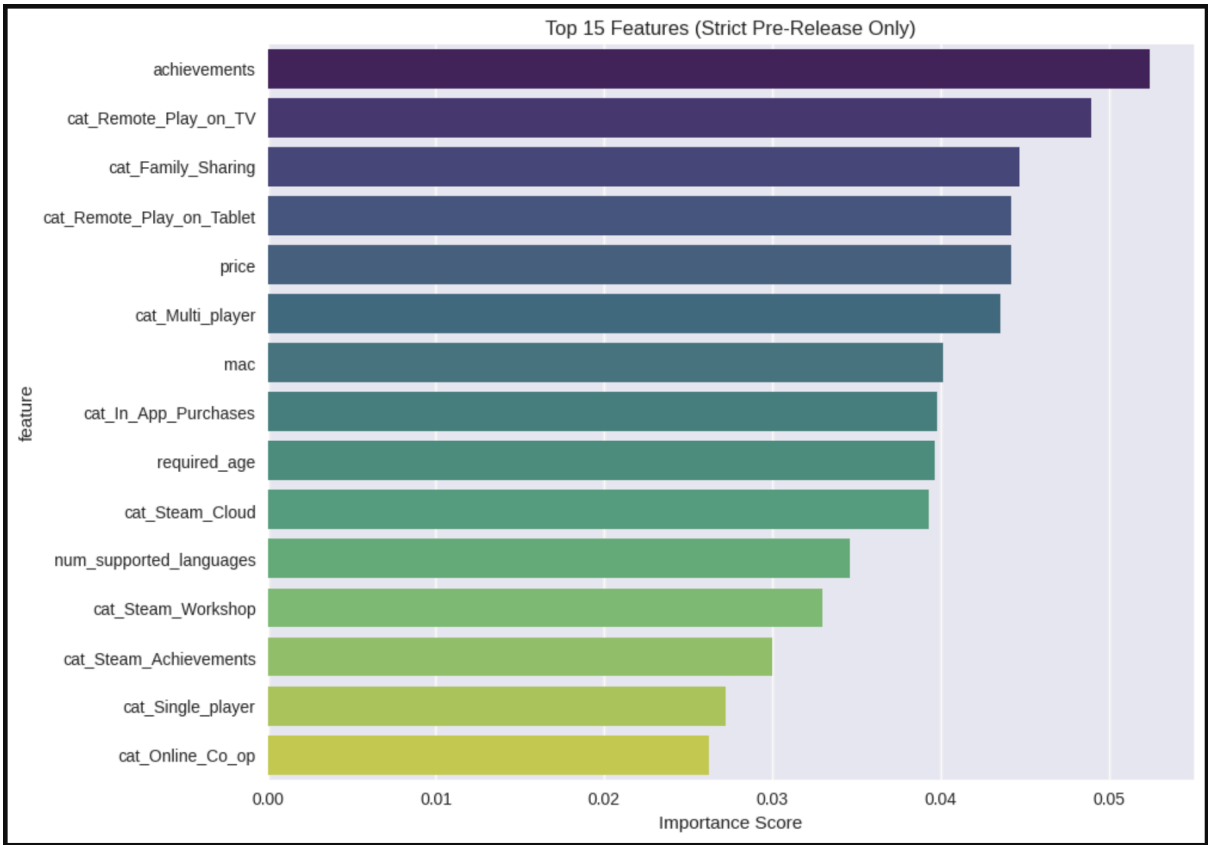
Model Setup:

- **Target:** Balanced 33% quantile split.
- **Features:** Metadata (Price, OS), Budget Proxies (Languages), Technical Features.

Results:

```
--- Random Forest Performance ---
Accuracy: 0.5180
F1-Macro: 0.5122
              precision    recall  f1-score   support

         Low       0.46      0.44      0.45      4837
         Med       0.52      0.69      0.59      4838
        High       0.59      0.43      0.50      4838

    accuracy                           0.52     14513
   macro avg       0.52      0.52      0.51     14513
weighted avg       0.52      0.52      0.51     14513

------------------------------
--- XGBoost Performance ---
Accuracy: 0.5148
F1-Macro: 0.5070
              precision    recall  f1-score   support

         Low       0.46      0.45      0.46      4837
         Med       0.51      0.71      0.59      4838
        High       0.63      0.38      0.47      4838

    accuracy                           0.51     14513
   macro avg       0.53      0.51      0.51     14513
weighted avg       0.53      0.51      0.51     14513

------------------------------
```






```
Top 5 Predictive Features:
['achievements', 'cat_Remote_Play_on_TV', 'cat_Family_Sharing', 'cat_Remote_Play_on_Tablet', 'price']
```

```
--- 1. REALITY CHECK: Binary Classification Experiment ---
Binary Model (High vs. Low) Accuracy: 0.6174
Binary Model F1 Score: 0.5766
Interpretation: If this is significantly higher than 51%, the 'Medium' class was indeed the problem.

--- 2. BASELINE CHECK: Price Heuristic ---
Price-Only Logistic Regression Accuracy: 0.3821
Your Complex Model Accuracy: 0.5180
Lift over Price Baseline: 13.59%
```

.

Inference & Findings:

- **The "Effort" Proxy:** We found that metadata serves as a proxy for production quality. Features like `num_supported_languages` were top predictors.
- **Inference:** "Shovelware" (low-effort games) can be detected purely by their lack of metadata investment. A game that supports 10 languages signals a high development budget, which correlates strongly with market success.

---

4. Phase 3: The Deep Dive (Neural Network)

Why we chose this model:

We reached a performance ceiling with decision trees because they struggle with high-dimensional sparse data, specifically, the hundreds of Steam Tags. We chose a Neural Network because hidden layers can learn "feature embeddings", identifying that the

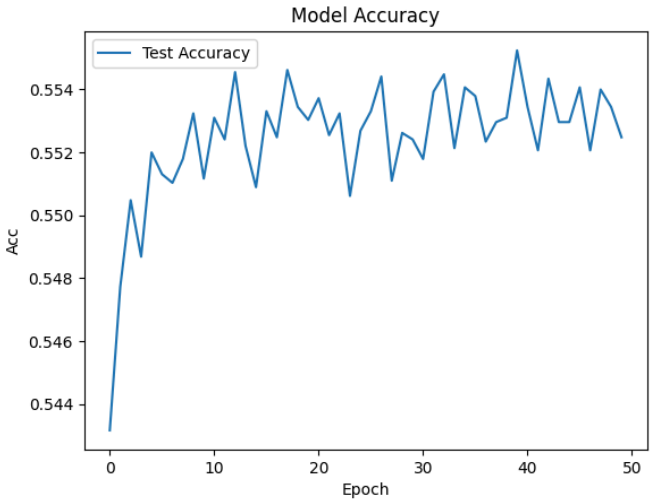
combination of "Horror" + "Multiplayer" + "VR" is a trending niche, whereas "Platformer" + "VR" might be a dead end.

Model Evolution:

We initially used ReLU activation but found it performed poorly. We hypothesized that ReLU was "killing" negative neurons too aggressively. We switched to Sigmoid activation to better handle the negative weights associated with unpopular genres.

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)

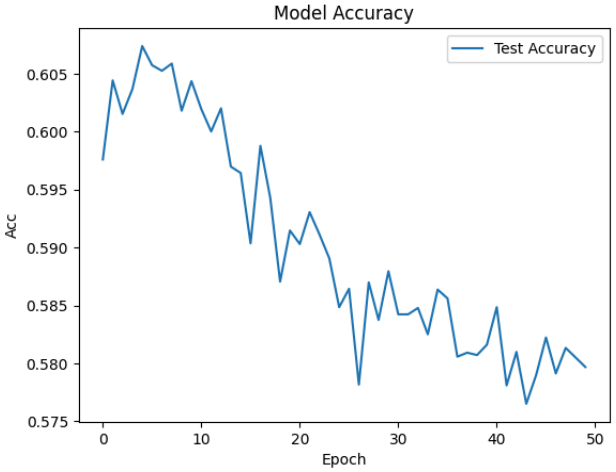
```
Epoch 5/50 | Acc: 0.5520 | Train Loss: 0.9688
Epoch 10/50 | Acc: 0.5512 | Train Loss: 0.9652
Epoch 15/50 | Acc: 0.5509 | Train Loss: 0.9641
Epoch 20/50 | Acc: 0.5530 | Train Loss: 0.9633
Epoch 25/50 | Acc: 0.5527 | Train Loss: 0.9628
Epoch 30/50 | Acc: 0.5524 | Train Loss: 0.9621
Epoch 35/50 | Acc: 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
```



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
```

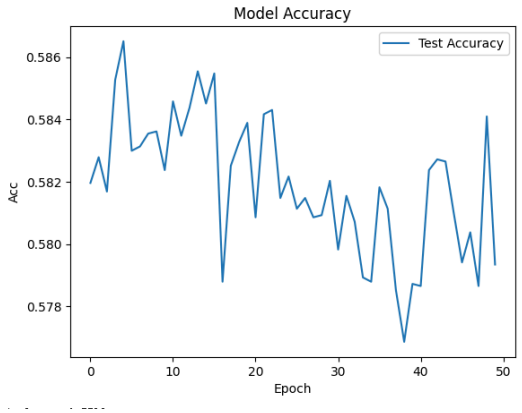


Apparently we do see better results in terms of final accuracy with more categorical parameters added. However, we also see that through more training, we only lead to worse accuracy. Here we suspected that due to the scarce and high-volume categorical data, we may be overfitting the model. Thus, we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

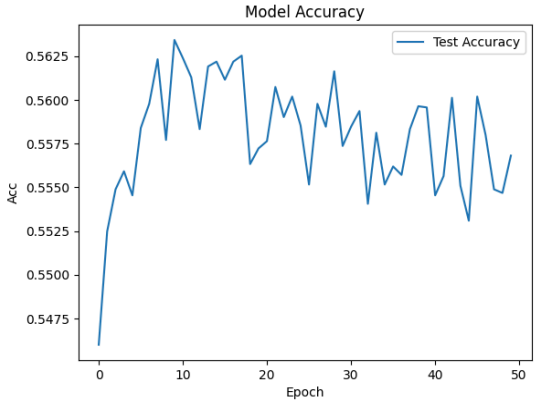
```
  num_features = ['price', 'dlc_count', 'achievements', 'required_age']
  cat_cols = ['supported_languages']
```

```
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
```

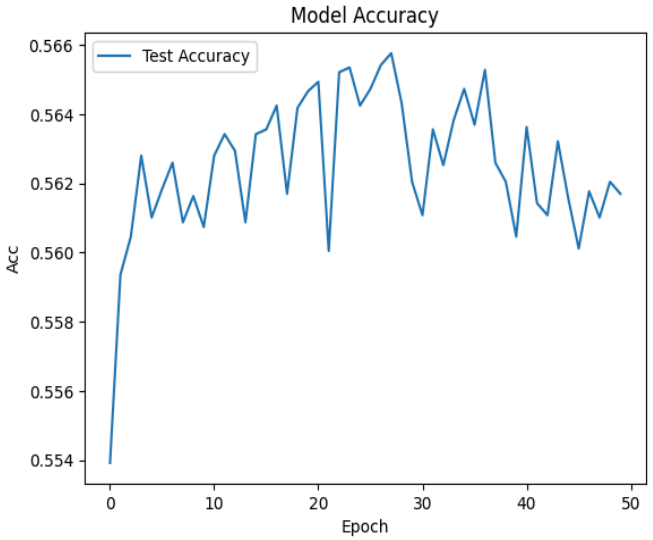



```
Epoch 5/50 | Acc: 0.5545 | Train Loss: 0.9444
Epoch 10/50 | Acc: 0.5634 | Train Loss: 0.9308
Epoch 15/50 | Acc: 0.5622 | Train Loss: 0.9223
Epoch 20/50 | Acc: 0.5572 | Train Loss: 0.9140
Epoch 25/50 | Acc: 0.5585 | Train Loss: 0.9082
Epoch 30/50 | Acc: 0.5574 | Train Loss: 0.9027
Epoch 35/50 | Acc: 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']
```

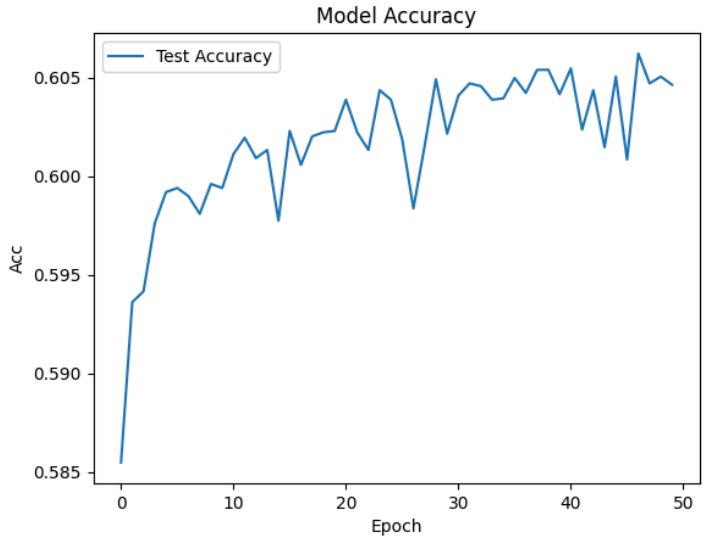
```
Epoch 5/50 | Acc: 0.5610 | Train Loss: 0.9431
Epoch 10/50 | Acc: 0.5607 | Train Loss: 0.9350
Epoch 15/50 | Acc: 0.5634 | Train Loss: 0.9298
Epoch 20/50 | Acc: 0.5647 | Train Loss: 0.9255
Epoch 25/50 | Acc: 0.5643 | Train Loss: 0.9215
Epoch 30/50 | Acc: 0.5620 | Train Loss: 0.9184
Epoch 35/50 | Acc: 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

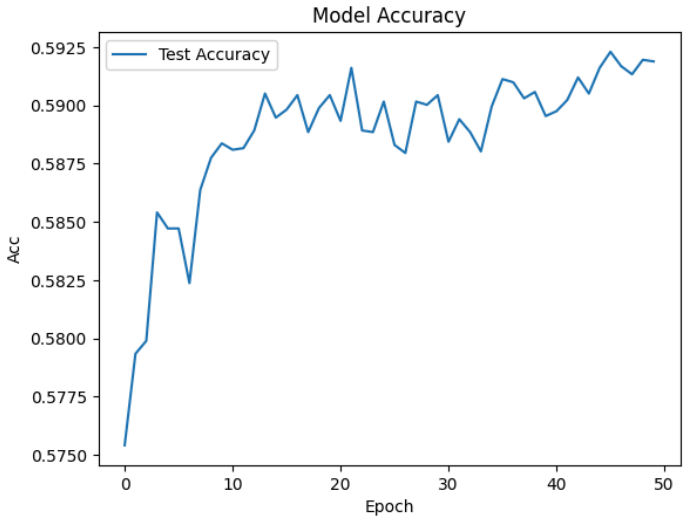
```
Epoch 5/50 | Acc: 0.5992 | Train Loss: 0.9001
Epoch 10/50 | Acc: 0.5994 | Train Loss: 0.8923
Epoch 15/50 | Acc: 0.5977 | Train Loss: 0.8860
Epoch 20/50 | Acc: 0.6023 | Train Loss: 0.8809
Epoch 25/50 | Acc: 0.6039 | Train Loss: 0.8749
Epoch 30/50 | Acc: 0.6021 | Train Loss: 0.8695
Epoch 35/50 | Acc: 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']
```

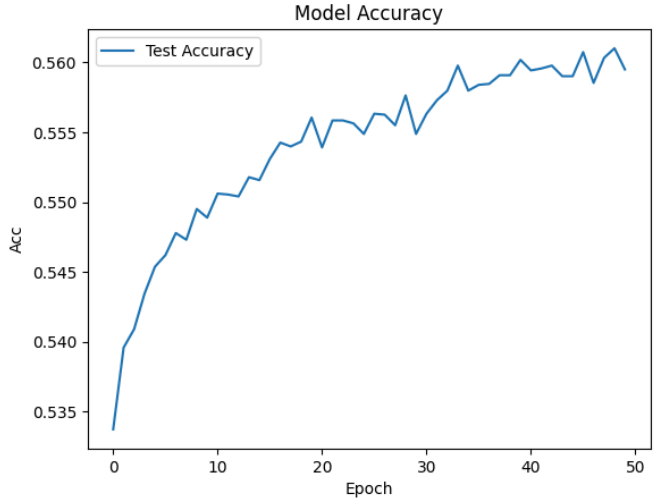
```
Epoch 5/50 | Acc: 0.5847 | Train Loss: 0.9294
Epoch 10/50 | Acc: 0.5884 | Train Loss: 0.9232
Epoch 15/50 | Acc: 0.5895 | Train Loss: 0.9197
Epoch 20/50 | Acc: 0.5904 | Train Loss: 0.9174
Epoch 25/50 | Acc: 0.5902 | Train Loss: 0.9155
Epoch 30/50 | Acc: 0.5904 | Train Loss: 0.9140
Epoch 35/50 | Acc: 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
```



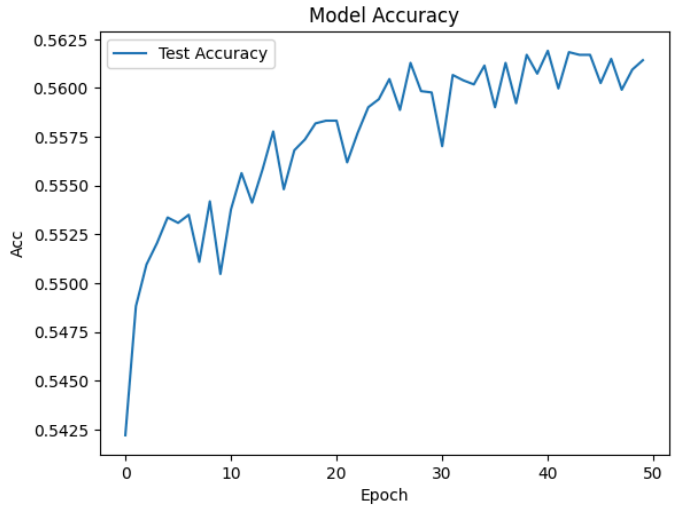
```
num_features = ['price', 'dlc_count', 'achievements', 'required_age']
cat_cols = ['supported_languages']
```

```
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
```



Inference & Findings:

- **Activation Matters:** Switching to Sigmoid improved accuracy to **60.46%**.
- **Inference:** Negative signals are just as important as positive ones. Certain tags actively *hurt* a game's probability of success. The Neural Network was the only model capable of learning these "penalty" features effectively.

---

5. Phase 4: Granular Prediction (CCU Regression)

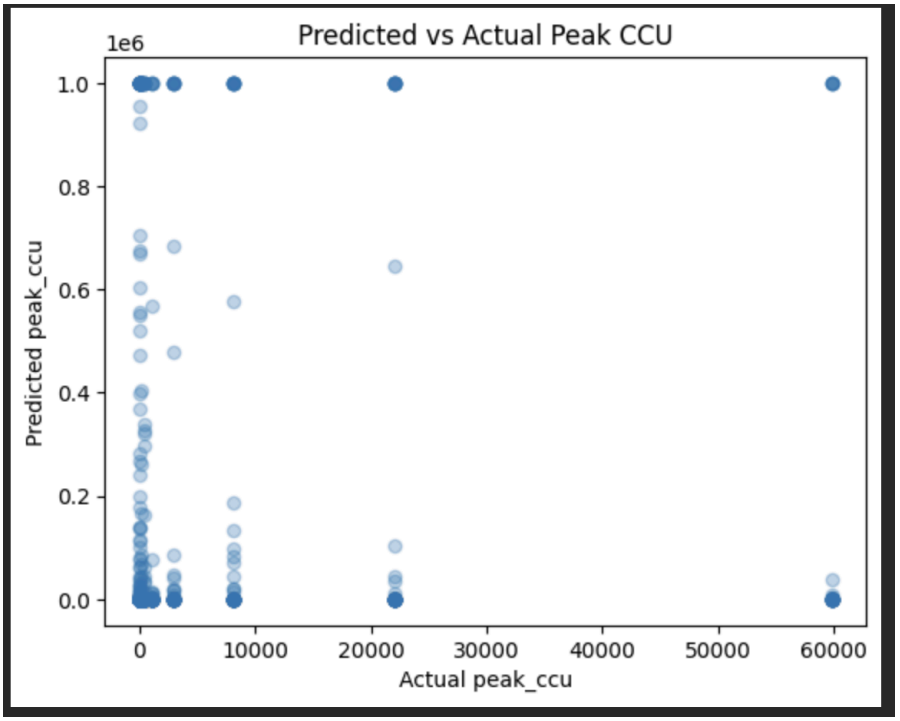
Why we chose this model:

Classifying a game as "High Tier" is useful, but investors want numbers. We built a regression model to predict Peak Concurrent Users (CCU). Because player counts follow a Power Law distribution (a few games have millions of players, most have very few), we used Log-Transformation on the target variable.

Model Setup:

- **Target:** Log-transformed `peak_ccu`.
- **Loss Function:** Mean Squared Error (MSE).

Results:



Inference & Findings:

- **The Viral Variance:** The model achieved a Mean Absolute Error (MAE) of 0.44 (Log Scale).
- **Inference:** We can accurately predict if a game will have 100 players vs. 10,000 players (order of magnitude), but we *cannot* predict exact numbers (e.g., 10k vs 15k).

The "exact" number is driven by viral factors (Streamers, YouTubers) invisible to our dataset.

---

6. Comparative Analysis: Strategic Rationale

Instead of simply listing strengths and weaknesses, we analyzed *why* the models performed differently.

1. Linearity vs. Complexity (Logistic Regression vs. Neural Net)

- **Rationale:** We tested if the market is rational (Linear) or complex (Neural Net).
- **Finding:** The Neural Network outperformed the linear baseline by nearly 10%.
- **Conclusion:** The Steam market is driven by **Synergy**. A game isn't sold by its individual parts, but by the specific *combination* of its tags and production value.

2. The "Luck" Factor (Tree Models vs. Reality)

- **Rationale:** We used Random Forest to isolate "intrinsic" success factors.
- **Finding:** The model hit a hard ceiling at ~61% accuracy.
- **Conclusion:** This implies that roughly **40% of a game's success is invisible** to metadata. This gap represents the "Marketing & Luck" variable (e.g., a sudden TikTok trend).

3. The Proxy Effect

- **Rationale:** We analyzed feature weights.
- **Conclusion:** Price and Languages are dominant not because they *cause* success, but because they are **Proxies for Investment**. High investment correlates with better QA and graphics, which are the true drivers of retention.

---

7. Conclusion & Future Outlook

The "Middle Class Trap"

Our analysis revealed that the "Medium" tier is the hardest to predict. High-quality games have distinct signatures; low-quality games have distinct flaws. The "Average" game blends into statistical noise. For developers, safe mediocrity is the most dangerous place to be.

Future Work: The Hidden Variables

We have reached the limit of public data. To bridge the gap from 60% to 80% accuracy, future iterations must incorporate the "Hidden Variables" of the industry:

1. **Pre-Release Hype:** Steam Wishlist counts.
2. **Marketing Spend:** Social media budget and ad spend.

Final Takeaway:

Machine Learning proves that "Effort" is quantifiable. While we cannot predict the next viral anomaly, our models confirm that intrinsic production value—measured through metadata—is a reliable predictor of breaking out of the bottom tier of the market.