STEAM REVIEW RADAR

START

ANDREW WANG CODY JIANG DYLAN ZHOU

HOW TO PLAY

THERE ARE 5 PARTS OF
OUR PRESENTATION!

- DATASET INTRO
- DATA PREPROCCESSING
- EDA & FEARTURES
- SPAM DETECTION
- CONCLUSIONS

PROBLEM MOTIVATION

Steam hosts millions of reviews → critical signal for recommendations and shoppers

But fresh reviews initially have no helpfulness votes, and platform is vulnerable to spam & duplicates.

We want to answer:

- which new reviews will be helpful?
- who are the reviewers behind them?
- how do we detect spam/near-duplicate reviews?

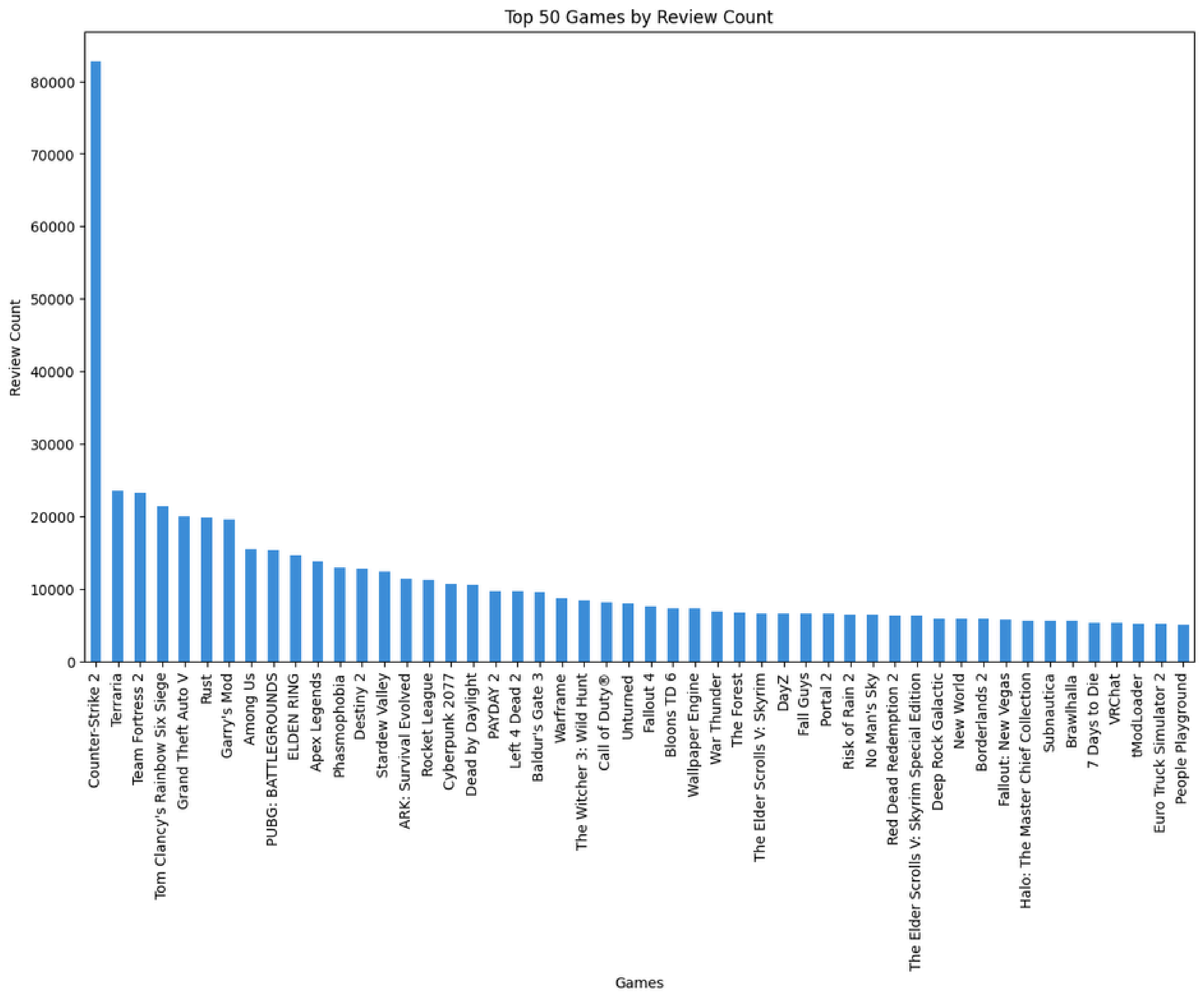
DATASET INTRODUCTION

Dataset Source

- Kaggle: 113,883,717 reviews (~42GB)
- Stored in CSV → Parquet
- Sample extraction sizes:
  - Small ≈ 20k for prototyping
  - Medium ≈ 2M for model training

Fields used

- Review text, helpful votes, funny votes
- Author playtime, ownership, purchase flags
- Timestamps, language, metadata



DATA PREPROCESSING PIPELINE

Processing Steps

1. Chunk-read CSV → downsample to reduce load of next steps
2. Filter for English reviews only
3. Probabilistic sampling → medium (2M) & small (20K) subsets
4. Remove unnecessary columns
5. Convert and store Parquet

Convert to Parquet

Sanity check for our new datasets and filter out some columns, then convert to parquet

```
sml_path = "steam_dataset/sml_sample.csv"
sml = pd.read_csv(sml_path)
sml = sml.drop(['hidden_in_steam_china', 'steam_china_location'], axis='columns', errors='ignore')

print(len(sml))
# sml.head()
# sml.columns
print(sml.dtypes)

sml.to_csv(sml_path, index=False, quoting=csv.QUOTE_NONNUMERIC)
```

```
20003
recommendationid                  int64
appid                             int64
game                             object
author_steamid                    int64
author_num_games_owned            int64
author_num_reviews                int64
author_playtime_forever           int64
author_playtime_last_two_weeks    int64
author_playtime_at_review         int64
author_last_played                int64
language                         object
review                           object
timestamp_created                 int64
timestamp_updated                 int64
voted_up                          int64
votes_up                          int64
votes_funny                       int64
weighted_vote_score             float64
comment_count                     int64
steam_purchase                    int64
received_for_free                 int64
written_during_early_access       int64
dtype: object
```

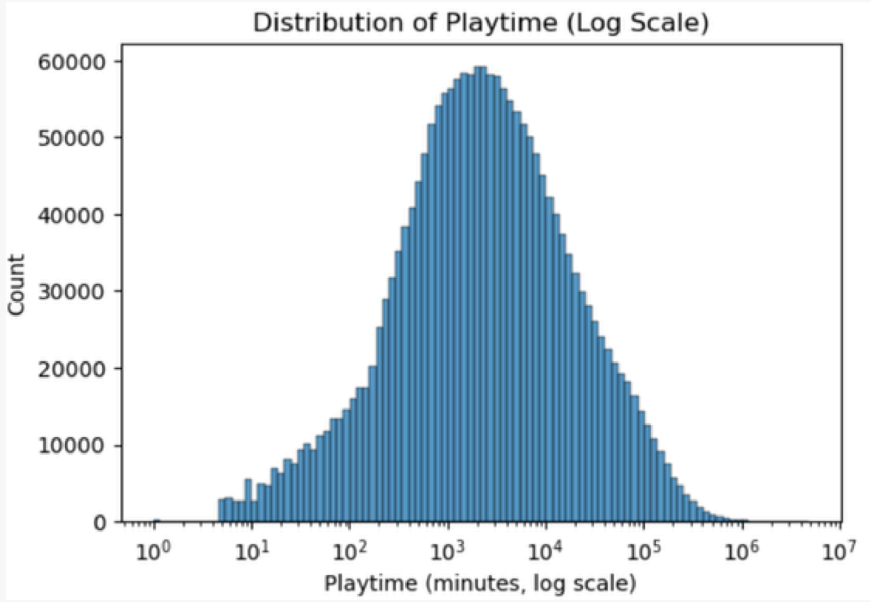
EDA HIGHLIGHTS & FEATURE ENGINEERING

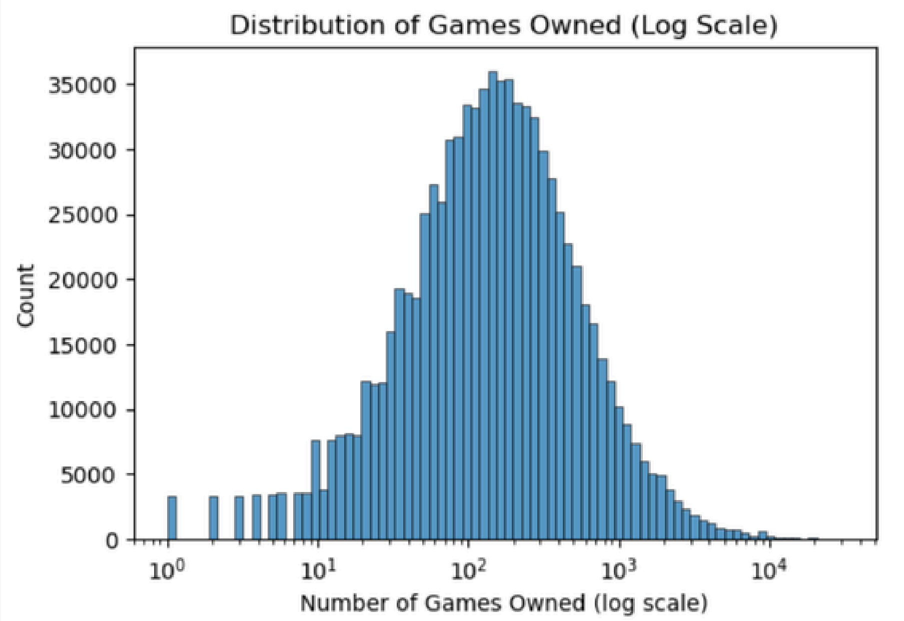
Key Observations

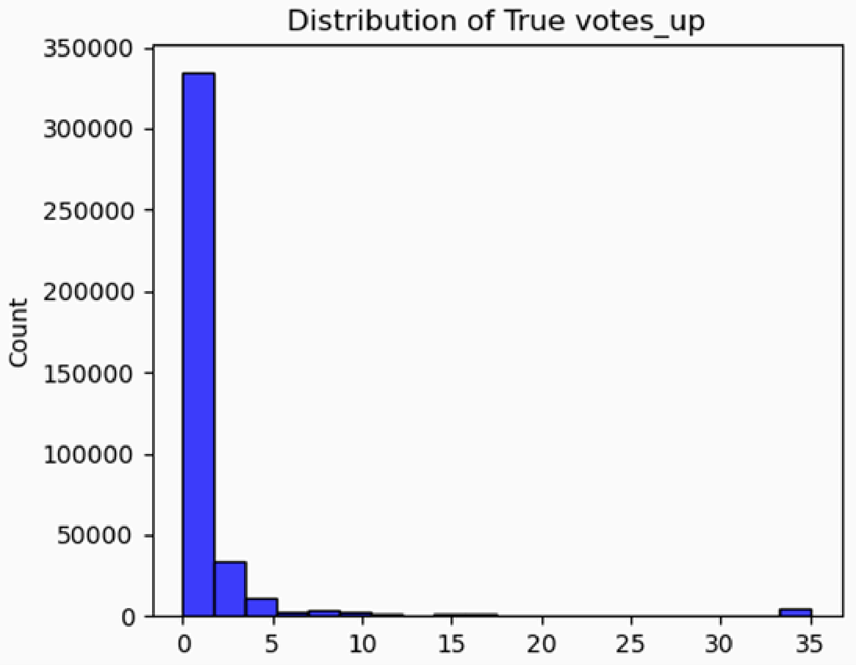
- Playtime & review count distribution heavily right-skewed
- Purchased reviews dominate; early-access is minority
- Many short reviews ("good", "nice") → low signal density

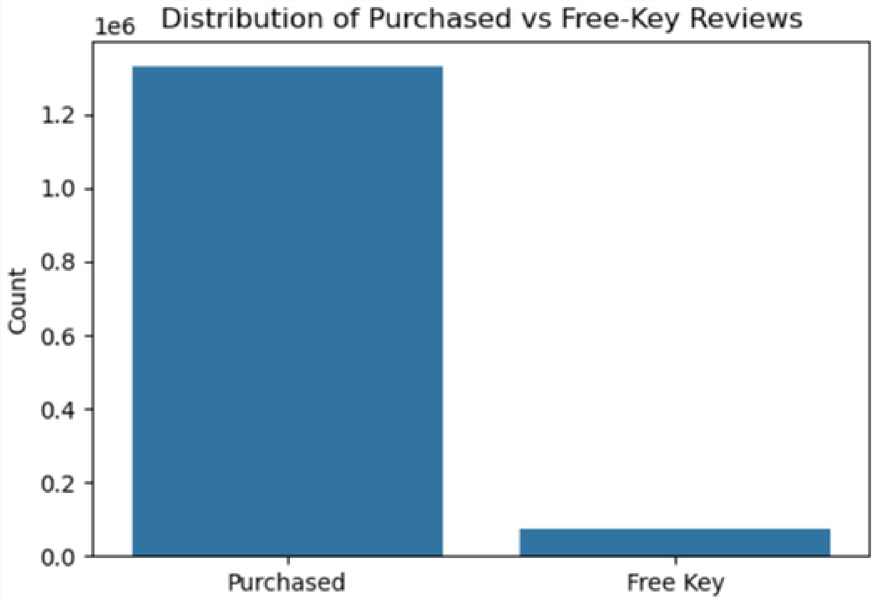
Features for model

📌 Text: TF-IDF (1–2 grams, 3k vocab)
📌 Numeric: playtime, reviewer history, funny upvotes…
📌 Categorical: language, purchase/early-access flags
📌 Target: votes_up (clipped top 1% to reduce long tail)





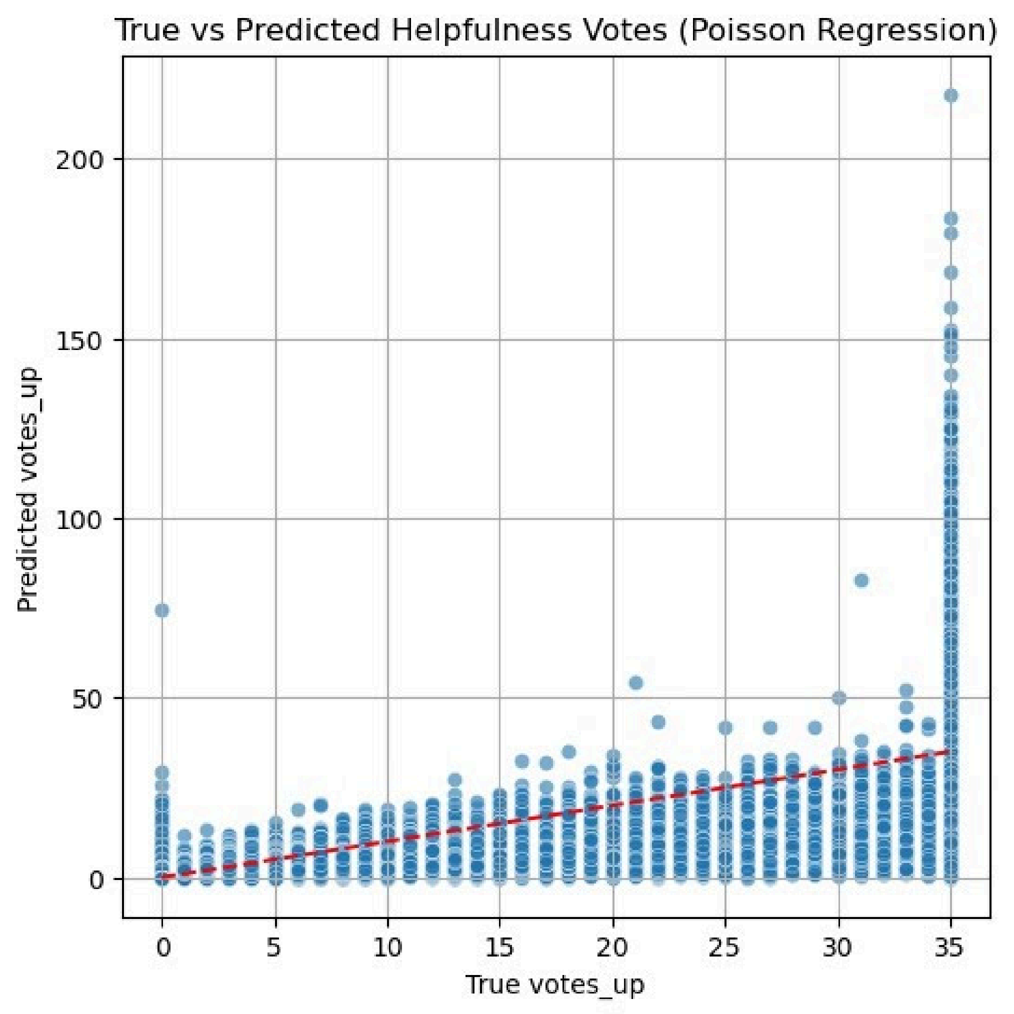


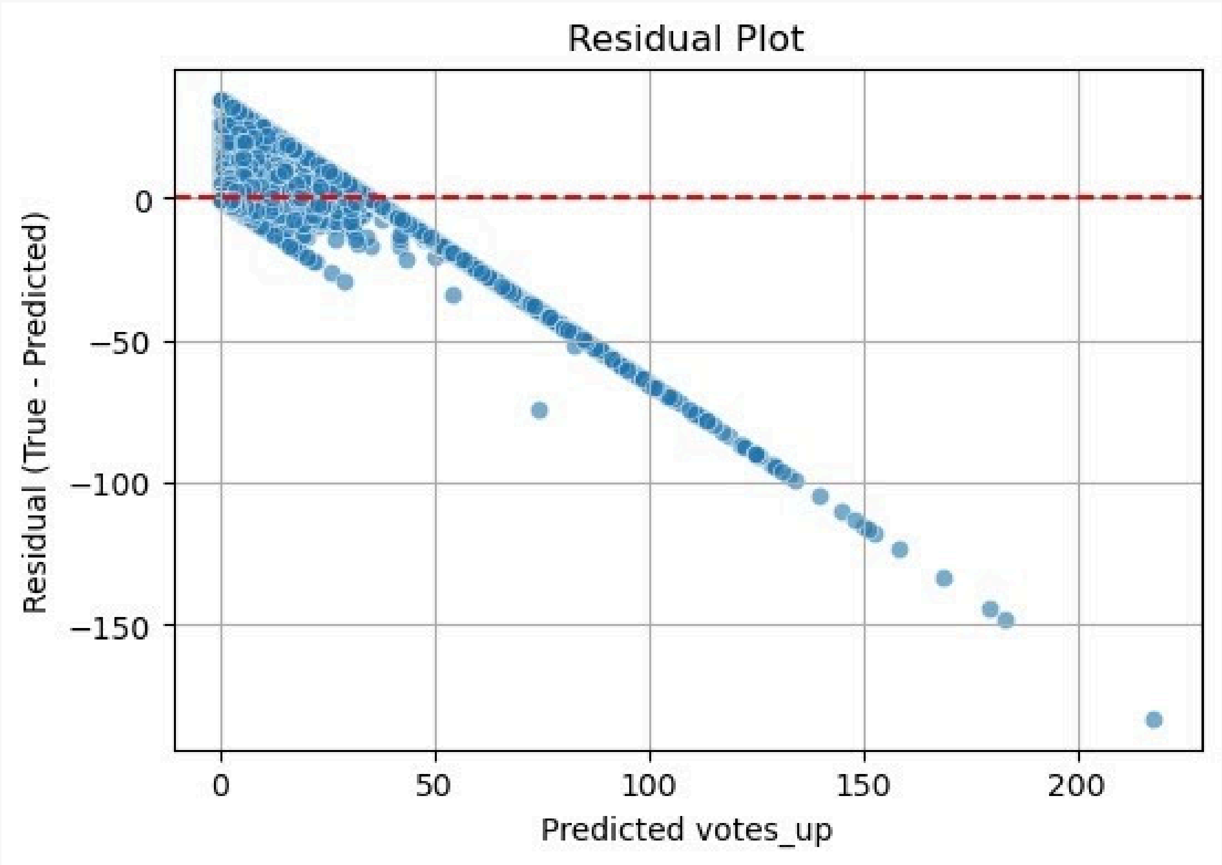


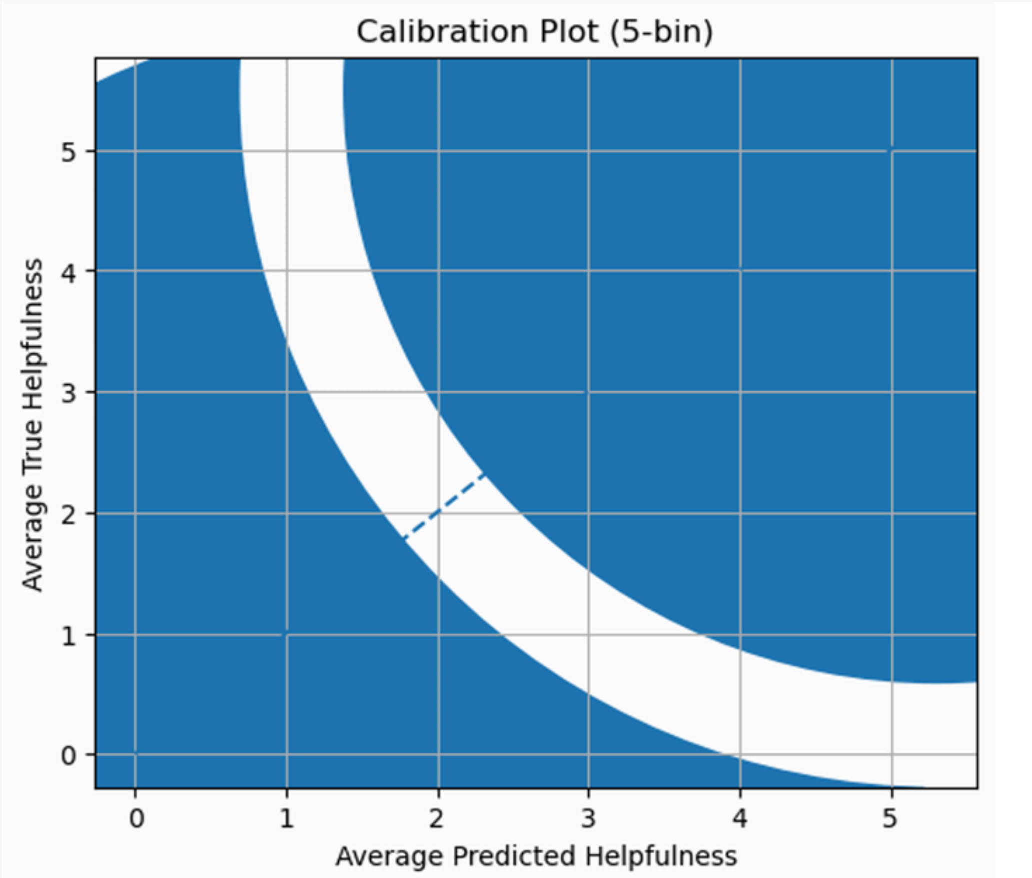
HELPFULNESS PREDICTION (MODEL RESULTS)

Model
- Poisson Regression pipeline
- Inputs: TF-IDF + metadata + numeric behavior
- Output: predicted helpful upvotes

Performance
- MSE = 10.16
- RMSE = 3.18
- MAE = 1.00
- Spearman ρ = 0.716 ← strong ranking correlation



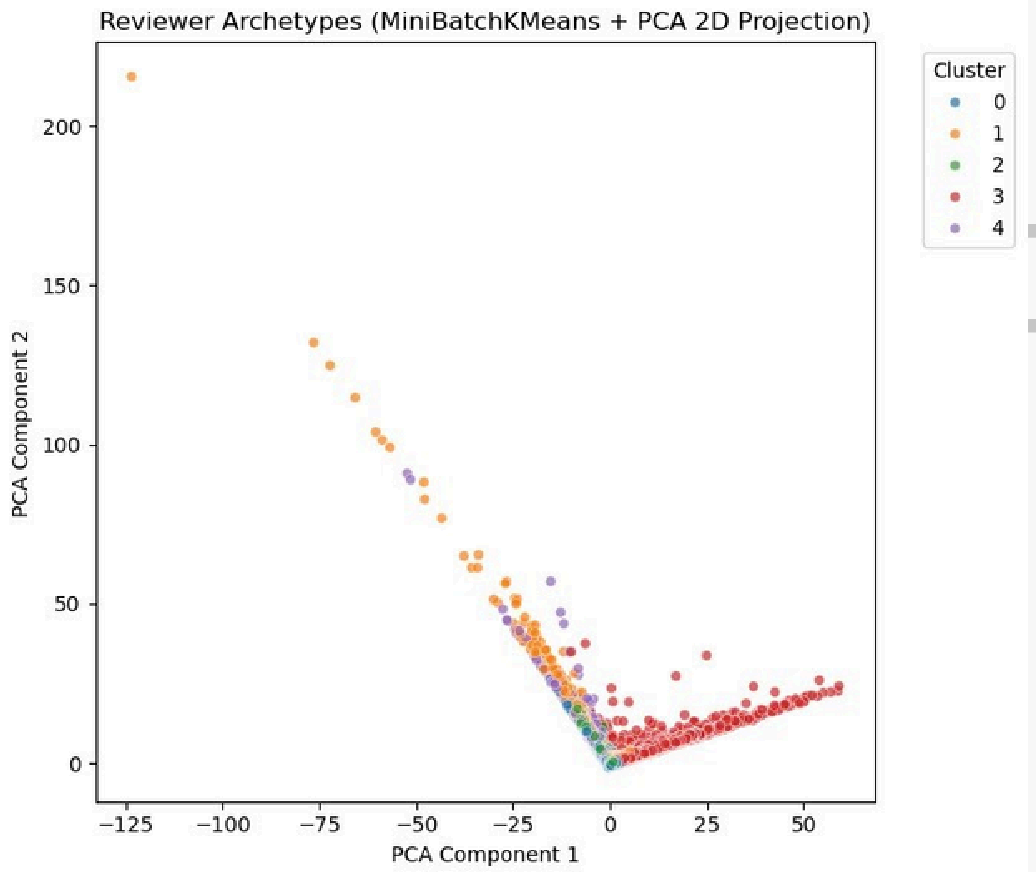




REVIEWER ARCHETYPES

Workflow

- Aggregate review → reviewer-level profile
- Standardize numeric metrics
- Clustering using MiniBatch K-Means (k=5)
- Compare behavioral differences



The Five Steam Reviewer Archetypes

The Influential Critic (Cluster 4)

- A veteran with a vast game library and high review count. Tends to leave critical/negative reviews that are recognized as **highly influential** (#007ACC), receiving the most helpfulness votes (avg. 6.97).

The Hardcore Specialist (Cluster 3)

- Owns a smaller library but has **extremely deep** (#007ACC) playtime (avg. 231k). Rarely reviews, but when they do, their reviews receive moderately high helpfulness.

The Free-Key Promoter (Cluster 1)

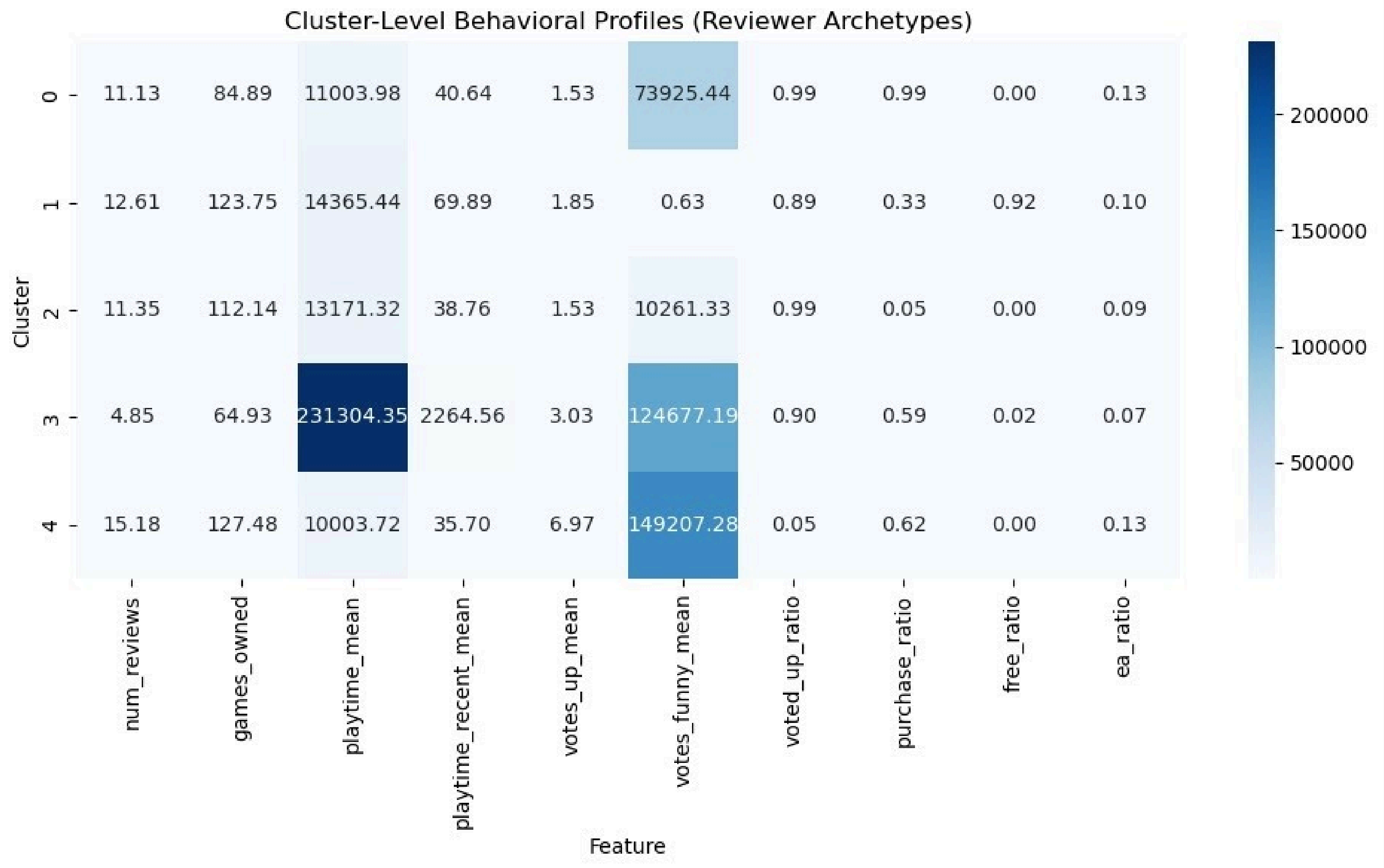
- A regular reviewer with a large library. Almost exclusively reviews games received for free (free_ratio of 0.92) and tends to write very positive reviews.

The Positive Purchaser (Cluster 0)

- A frequent reviewer with a sizable library who mostly reviews purchased games. They are highly positive but their reviews typically receive low community helpfulness.

The Mainstream Reviewer (Cluster 2)

- Similar to the Positive Purchaser but with more mixed purchase/free behavior. Owns many games, reviews regularly, and is very positive, but also receives low helpfulness.



SPAM DETECTION PIPELINE ---- WHY WE NEEDED SPAM DETECTION

Steam has millions of user voices — but not all are real signals.
We noticed many reviews were one-liners, templated, or repeated across games.
Those reviews artificially boost helpfulness metrics and pollute training.
If we train on them directly → our helpfulness model learns wrong behaviors.

Our Questions

- How much of the review ecosystem is copypasta / near-duplicate?
- Can we automatically flag suspicious reviews at scale?
- What if we remove or down-weight them — do predictions improve?

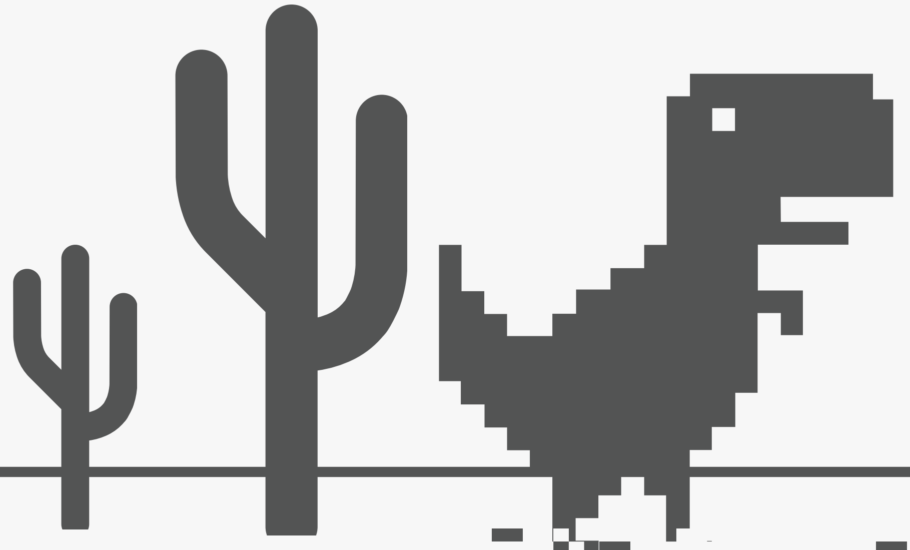
Goal

→ Build a lightweight, scalable spam radar that runs on millions of reviews
→ Without deep neural embeddings or heavy compute

```
======================
Pair: 899 vs 12454

[Review A]:
good

[Review B]:
good
======================
Pair: 1829 vs 18023

[Review A]:
great game

[Review B]:
great game
```

[8]:

	review_clean	word_count	unique_ratio	avg_word_len
0	thumbsup	1	1.000000	8.000000
1	there s like no guidance on what to do i got a...	46	0.847826	3.239130
2	2d minecraft never felt so good	6	1.000000	4.333333
3	the battlepass is heresy	4	1.000000	5.250000
4	oooooo	1	1.000000	6.000000



SPAM DETECTION PIPELINE ---- HOW WE DETECT SPAM

We started small — cleaned 20k reviews first, so we could iterate fast.
We processed raw text into normalized tokens to compare reviews fairly.

1. Clean Text
   - lowercase, remove symbols, collapse whitespace
   - Example: ":thumbsup:" → "thumbsup"
2. Generate 5-gram shingles
   - Turn each review into overlapping character chunks
   - "great game" → ["great","reat ","eat g","at ga","t gam"," game"]
3. MinHash Signatures
   - Hash shingles into compact fingerprints
   - Makes large-scale similarity search efficient
4. LSH (Locality Sensitive Hashing)
   - Bucket reviews with similar fingerprints
   - Near-duplicate threshold ≈ 0.8 Jaccard
5. Output: Pairs of suspiciously similar reviews

Evidence of success

➤ On 19,770 reviews, 87,267 duplicate pairs found

```python
# Build MinHash signatures and LSH index
num_perm = 128

def build_minhash(shingles):
    """
    Build a MinHash signature for a list of shingles.
    """
    m = MinHash(num_perm=num_perm)
    for s in shingles:
        if s:  # skip empty strings
            m.update(s.encode("utf8"))
    return m

df = df.reset_index(drop=True)

minhashes = {}
for idx, shingles in df["shingles_5gram"].items():
    m = build_minhash(shingles)
    minhashes[idx] = m

print("Number of MinHash signatures built:", len(minhashes))
```

```
Number of MinHash signatures built: 19770
```

```python
# Build MinHash LSH index
threshold = 0.8

lsh = MinHashLSH(threshold=threshold, num_perm=num_perm)

for idx, m in minhashes.items():
    lsh.insert(str(idx), m)

print("LSH index built.")
```

```
LSH index built.
```

```python
# Query the LSH index to find near-duplicate pairs

pairs = set()

for idx, m in minhashes.items():
    candidates = lsh.query(m)
    i = idx
    for c in candidates:
        j = int(c)
        if j == i:
            continue
        a, b = sorted((i, j))
        pairs.add((a, b))

print("Unique near-duplicate pairs found:", len(pairs)

sample_pairs = list(pairs)[:10]
for (a, b) in sample_pairs:
    print("=" * 80)
    print(f"Pair: {a} vs {b}")
    print("\n[Review A]:")
    print(df.loc[a, "review_clean"])
    print("\n[Review B]:")
    print(df.loc[b, "review_clean"])
```

```
Unique near-duplicate pairs found: 87267
```

SPAM DETECTION PIPELINE ---- WHAT WE FOUND + WHY IT MATTERS

Once we flagged duplicates, we measured how different they actually were.
And the results confirmed what we suspected — spam reviews behave differently.

Key Findings

- 14.74% of reviews were near-duplicate
- Style anomaly (IsolationForest) caught ~2% unusual-writing reviews
- Spam flagged reviews had much lower helpful votes:

Group	Avg votes_up
Normal reviews	3.09
Near-duplicate spam	0.54
Style anomalies	17.8* (rare but extreme cases)

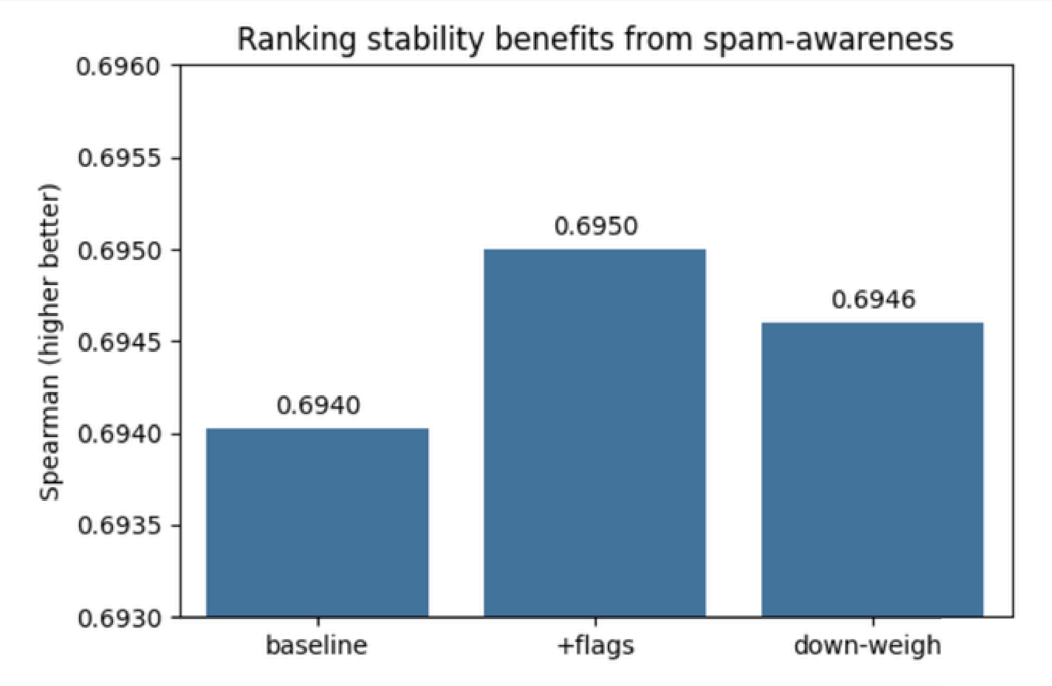
Why it matters

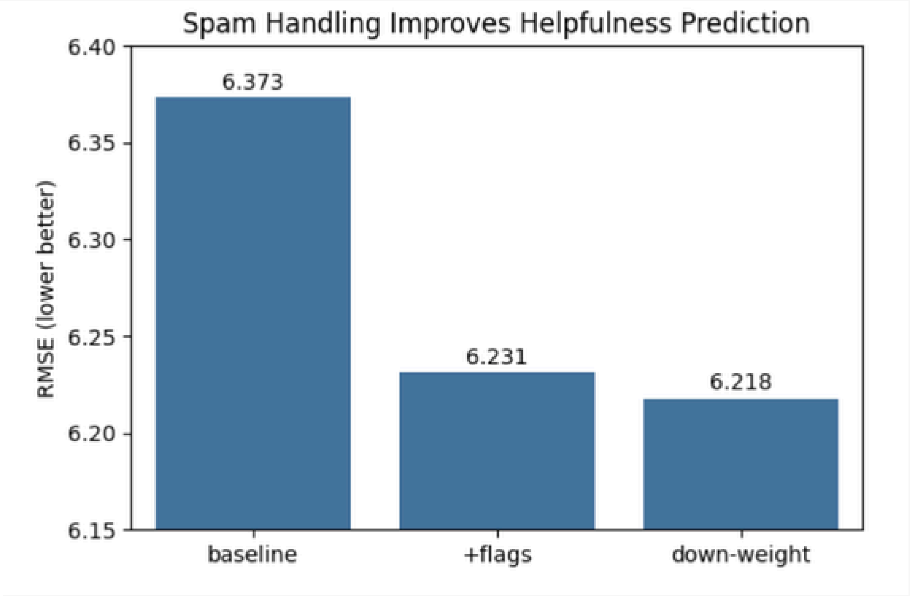
- Spam reviews drag down regression training
- We can drop them, down-weight them, or add flags as model features
- Even baseline Poisson regression improved when spam was removed

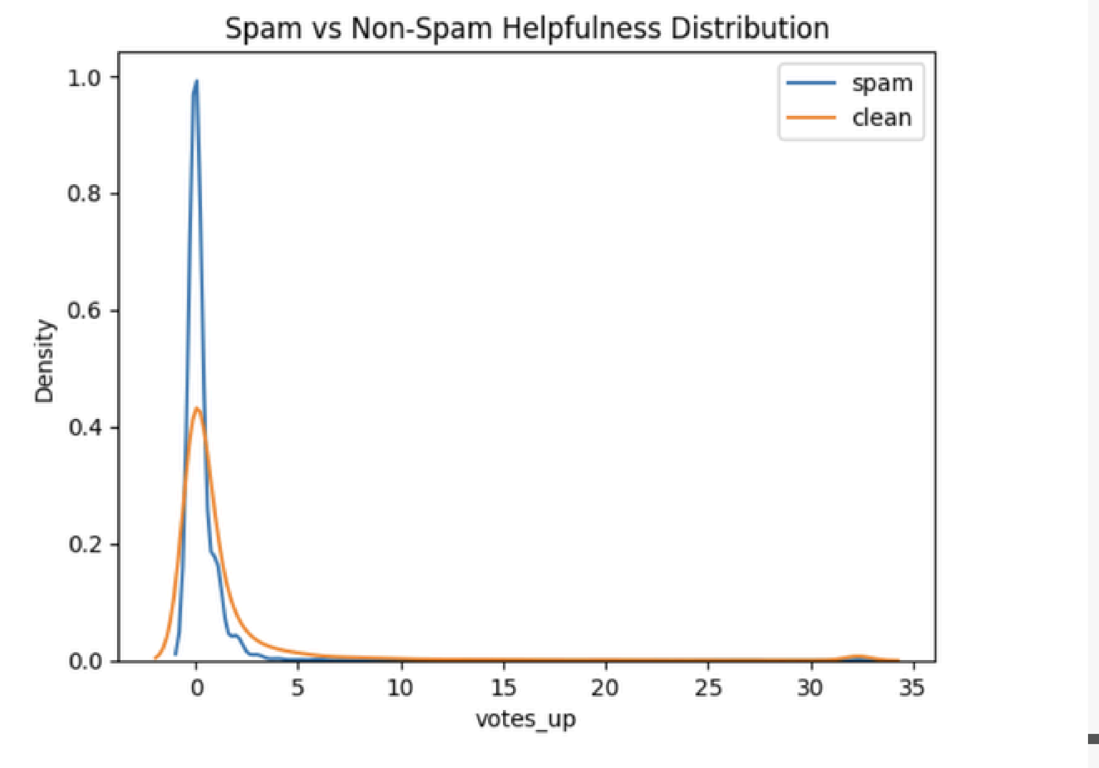
Future

- Use transformer embeddings for semantic copies
- Live spam radar could run in stream (Kafka → Spark Structured Streaming)

"By removing spam, we clean the signal.
By boosting real reviews, we support authentic player voices."







CONCLUSION & TAKEAWAYS

Achievements

- 71.6% rank correlation helpfulness predictor
- Reviewer archetypes reveal platform behavioral patterns
- 14.7% duplicate detection improves data quality

Future Improvements

- Train XGBoost Poisson / GBDT
- Incorporate semantic embeddings (BERT)
- Deploy streaming real-time scoring agent

THANK YOU!

QUESTIONS WELCOME 🎮