Movie Success & Revenue Prediction System – Final Project Report

Course: CIS 593 - Data Mining and Machine Learning

Student: Niteesh Singh **CSU ID:** 2886321
Student: Dharmik Kurlawala **CSU ID:** 2886995

1. Introduction & Project Goal:

Predicting a movie's success before its release is a challenging but valuable task in the film industry. This project aims to build a Movie Success Prediction system using AI/ML techniques on a rich dataset from TMDB (The Movie Database). In the context of data analytics and AI, our goal is to analyze historical movie data and train models to predict whether a movie will be a "success" or "flop" based on features available before release. A movie's success is defined as earning revenue greater than 1.5 times its budget, which indicates a significant return on investment. The project demonstrates how data-driven insights (from cast and crew information, genre, budget, etc. can be used to anticipate box office outcomes.

2. Dataset Description:

Data Sources & Collection: The dataset is a curated combination of two TMDB data sources:
Main Dataset (TMDB_movie_dataset_v11.csv): This contains core movie information for a broad set of films (e.g., title, release date, budget, revenue, runtime, genres, etc.). It is likely derived from a TMDB or Kaggle dataset of movies.
Secondary Dataset (TMDB_all_movies_1M.csv): This is an extended dataset (up to one million movie entries) providing additional details for movies, such as full cast lists, directors, writers, producers, cinematography, composer, and IMDb ratings/votes. It serves to enrich the main dataset with extra features by matching on movie IDs.
Merging Strategy: The two datasets were merged on the unique TMDB id field. The main dataset provides each movie's fundamental attributes, while the secondary dataset contributed extra columns like cast, director, writers, producers, imdb_rating, imdb_votes, etc., for each movie. We performed a left-join, keeping all movies from the main set and adding whatever extra info was available from the secondary set. Resulting Dataset Size: After merging, the enriched dataset contained thousands of movies (covering a wide range of years and genres). We then applied filters (see Preprocessing below) to focus on movies with complete data and meaningful values. After cleaning, the final filtered dataset has 8,835 movies, each with a comprehensive set of features. This filtered dataset (TMDB_filtered_labeled.csv) is the basis for feature engineering and model training. Features (Raw): Initially, for each movie we have:
ID, Title, Release Date – basic identifiers.
Budget and Revenue – in USD.
Runtime – movie duration in minutes.
Genres – list of genres (one movie can belong to multiple genres).
Spoken Languages – list of languages spoken in the movie.
Production Companies and Production Countries.
Cast – list of cast member names.
Director, Writers, Producers – principal crew names.
TMDB Vote Average and Count – user ratings on TMDB (post-release metrics).

IMDb Rating and Votes – user ratings on IMDb (post-release).
Overview – a brief synopsis/description of the movie.
Popularity – a TMDB popularity score (which can be influenced by post-release factors).
Other crew roles (cinematographer, composer) – present in secondary data but not all were ultimately used.
These features were collected from TMDB's database (and IMDb for the ratings) via the provided CSVs. The merging and cleaning steps ensured we have a single table (TMDB_movie_dataset_Full.csv) with all relevant columns per movie. Data Characteristics: The movies span various years (several decades), multiple languages, and all genres. The budget values range widely from small independent films (~thousands USD) to big blockbusters (hundreds of millions USD). Similarly, revenues range from near-zero to billions. The cast and crew lists vary in length per movie (from a few key people to dozens of names). This diversity required careful preprocessing to make the data usable for modeling.

```python
def read_table(path, usecols=None):
    ext = os.path.splitext(path)[1].lower()
    if ext in (".xls", ".xlsx"):
        return pd.read_excel(path, usecols=usecols)
    else:
        try:
            return pd.read_csv(path, usecols=usecols, encoding="utf-8")
        except UnicodeDecodeError:
            print(f"⚠️  UnicodeDecodeError reading {path}, retrying with latin-1")
            return pd.read_csv(path, usecols=usecols, encoding="latin-1")
```

3. Project Goal in Context:

The goal is to use the above dataset to train machine learning models that predict movie success. In a data analytics context, this involves:
Identifying which features of a movie (known before release) correlate with financial success.
Building a classifier that outputs whether a movie will be a success (box office hit) or a flop.
Optionally, providing a revenue prediction (regression) to estimate how much money the movie will gross.
From an AI application perspective, this project highlights how predictive modeling can assist decision-making in the entertainment industry. Studios and investors could use such a model to evaluate scripts or projects by inputting details (cast, director, budget, etc.) and get a probability of success. This is a demonstration of applying data science to a real-world scenario where outcomes are uncertain and multi-factorial. The project not only focuses on raw predictive accuracy but also on model interpretability and reliability:
By analyzing which features are most influential (e.g., "star power" of cast, budget size, genre trends).
By calibrating the model's predicted probabilities so they reflect true likelihoods of success.
By evaluating performance with multiple metrics (accuracy, precision/recall, ROC-AUC) to ensure the model is robust and fair to both classes (success and flop).
In summary, the goal is to turn historical movie data into an AI-driven insight: "Given the attributes of a new movie, can we predict if it will succeed or not?"

4. System / Tools Used:

This project was implemented using the Python programming language (Python 3.x) in a Jupyter Notebook environment. Key tools and libraries include:
Pandas (for data loading, merging, and manipulation).
NumPy (for numerical computations).
Scikit-Learn (sklearn) (for preprocessing, model splitting, metrics, and some model components like calibration).
TensorFlow & Keras (for building and training the Artificial Neural Network classifier).

XGBoost (for training the regression model to predict revenue).
Matplotlib and Seaborn (for data visualization, plotting graphs such as confusion matrix, ROC, PR curve, etc.).
TextBlob (for natural language processing, used to compute sentiment of the movie overview text).
MultiLabelBinarizer from sklearn (for encoding genres and languages, which are multi-value categorical features).
QuantileTransformer from sklearn (for scaling/normalizing numerical features in a robust way).
Joblib (for saving and loading trained models and preprocessed artifacts like feature lists, scalers).
Flask (though not the focus of this report, a Flask app was used to deploy the model – as evidenced by form.html and API_3.py – enabling a web interface to input movie features and get predictions).
Development Environment: The code was run on a standard PC (Windows 10/11 or Linux) with Python 3.9. Libraries were installed via pipe. For example, to set up the environment, one would need to run:
pip install pandas numpy scikit-learn tensorflow keras xgboost textblob unidecode matplotlib seaborn
Additionally, NLTK corpora or TextBlob data might be needed for sentiment analysis (TextBlob typically requires downloading the punkt tokenizer and sentiment lexicons). This can be done by installing textblob and running python -m textblob.download_corpora. The platform used was Jupyter Notebook for iterative development, and results were exported/visualized within that environment. The final code was also structured into Python scripts for reproducibility:
Dataset_Creation.py (for merging datasets),
Filtered_Columns.py (for initial filtering and labeling),
Data_Processing.py (for feature engineering),
Train_Model_3.py (for model training),
Evaluate_Model_3.py (for evaluation and plotting).
Hardware: The model training (especially the ANN and XGBoost) was not extremely computationally heavy given the dataset size (~8.8k samples). A modern laptop/desktop CPU (and optionally a GPU for the neural network) is sufficient. Training the neural network for 60 epochs and performing 5-fold cross-validation for the XGBoost took on the order of minutes.

5. Feature Selection Methodology

Selecting the right features is crucial for predictive performance and generalization. In this project, feature selection was driven by domain knowledge and data availability (rather than automated feature elimination techniques). The guiding principles were:
Include features that are known before release (to avoid data leakage from future information).
Cover various aspects that might influence a movie's success: financial, temporal, content, and personnel.
Leverage the rich data by creating new features (feature engineering) that condense useful information (e.g., "star power" metrics from cast/crew histories).
Exclude features that are redundant or strongly correlated with others, to simplify the model.
Initial Features Considered: All columns listed in the dataset description were considered. However, some of these are post-release indicators (for example, vote_average, vote_count, popularity, IMDb ratings) which would not be realistically available before a movie comes out. Including them would introduce target leakage (because a movie that is known to have a high IMDb rating is likely already successful). Therefore, we excluded all post-release popularity and rating metrics from the predictive features. Feature Engineering & Selection: We crafted several new features:
"Star Power" Features: We created metrics to quantify the influence or past success of key entities:
cast_power – measures the combined box-office clout of the top cast members.
director_power, writers_power, producers_power – similar metrics for director and other creators.
production_companies_power, production_countries_power – metrics reflecting past successes of the studios and countries involved.
How star power is computed: For each person or entity (actor, director, etc.), we aggregated their past movie performances (cumulative profits, weighted ratings, popularity) over time. Then for a given new movie (with release year Y), we look up each cast/crew member's aggregated success up to year Y-1. For cast, we specifically take the

top 3 actors (assuming the leading actors have the most impact) and combine their scores to get the movie's cast_power. For a director or others, which is usually a single person, we use that person's score. These scores effectively represent how "proven" the talent behind the movie is, based on historical data.

Sentiment of Overview: overview_sentiment – we applied sentiment analysis to the movie's plot synopsis (overview text). Using TextBlob's sentiment polarity, we get a value between -1 (very negative sentiment) to +1 (very positive). The hypothesis is that movies with certain sentiment profiles in their description (e.g. very uplifting or very poignant descriptions) might correlate with audience appeal. We clipped the sentiment values to a range [-0.5, 0.5] to reduce the impact of outliers/extremes and to keep the scale moderate.

Release Time Features:
release_year – the year of release.
release_month – month of release.
is_holiday_release – a binary flag indicating if the movie was released in November or December, which is often the holiday/award season when big movies are launched. Holiday releases might have an advantage in audience turnout.

Sequel Indicator: is_sequel – a binary feature set to 1 if the movie's title suggests it, is a sequel. We looked for common sequel patterns in the title (e.g., having a "2" or "II", "Part II", etc. in the title). The rationale is that sequels of successful franchises often have a built-in audience and tend to perform well (though not always).

Major Studio Indicator: big_studio – a binary feature indicating if any of the production companies is a major, well-known studio (from a predefined list of top studios like Disney, Warner Bros, Universal, etc.). Major studios have more marketing power and distribution, often correlating with higher success rates.

Director Hit Ratio: director_hit_ratio – for each director, we computed the fraction of that director's past films in the dataset that were labeled "success". This gives a sense of how consistently that director makes hits. For a new movie, we take the director's hit ratio as a feature. A high ratio means the director has a track record of successes.
Genres (One-hot encoding): Movies often belong to multiple genres (Action, Comedy, Drama, etc.). We used a MultiLabelBinarizer to one-hot encode each genre as a binary feature (e.g., genre_action=1 if the movie has Action as one genre, else 0). This results in ~20 genre features (the dataset had genres like Action, Adventure, Animation, Comedy, Crime, Documentary, Drama, etc.). This allows the model to learn which genres (or genre combinations) tend to succeed (e.g., perhaps family animations have high success rates, etc.).
Spoken Languages (One-hot encoding): Similarly, the primary spoken languages of a movie were one-hot encoded. The dataset includes a wide range of languages (English, French, Spanish, etc. up to less common ones). We encoded all languages present in the data as binary features (lang_english, lang_french, ...). This turned into a large set of features (dozens of language flags). In practice, many movies have English as one language, so lang_english is very common. Language might correlate with market size or niche appeal (e.g., English or Mandarin films have access to bigger markets).

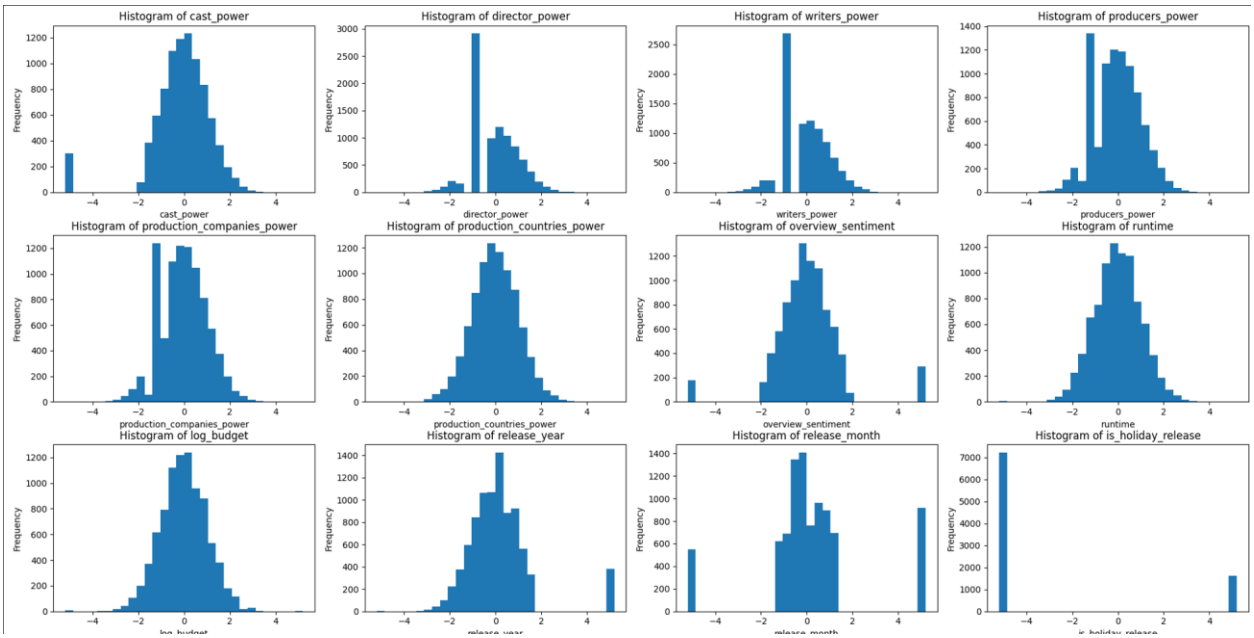
```python
# Multi-hot encodings
mlb_gen = MultiLabelBinarizer()
genre_df = pd.DataFrame(mlb_gen.fit_transform(df["genres"].apply(parse_list)),
                        columns=[f"genre_{g}" for g in mlb_gen.classes_])
joblib.dump(mlb_gen,"mlb_genre.pkl")

mlb_lang = MultiLabelBinarizer()
language_df = pd.DataFrame(mlb_lang.fit_transform(df["spoken_languages"].apply(parse_list)),
                           columns=[f"lang_{l}" for l in mlb_lang.classes_])
joblib.dump(mlb_lang,"mlb_lang.pkl")

tfidf = TfidfVectorizer(max_features=50)
tfidf_mat = tfidf.fit_transform(df["overview"].fillna("")).toarray()
tfidf_df = pd.DataFrame(tfidf_mat, columns=[f"tfidf_{i}" for i in range(tfidf_mat.shape[1])])
joblib.dump(tfidf,"overview_tfidf.pkl")

df["overview_sentiment"] = df["overview"].fillna("").apply(overview_sentiment)
df["overview_sentiment"] = df["overview_sentiment"].clip(-0.5, 0.5)
```

Log-Scaled Budget: log_budget – we use the natural log of (budget + 1) to scale down the huge range of budget values. Budget in dollars can range over 6-9 orders of magnitude, so taking log helps stabilize the variance and makes the feature less skewed.
(Note: We also compute log_revenue similarly for use in the regression model, but log_revenue is not used for classification— it is basically our target for regression.) After engineering, we reviewed the feature list and decided to exclude the following from input features:
The textual fields themselves (overview text, title text, etc.) are not directly used except via the engineered sentiment and sequel flag (using title) and the TF-IDF vector (using overview, see below).
TMDB/IMDb votes and ratings (as mentioned, they are post-release).
popularity (post-release metric on TMDB).
Redundant ID or name fields (we keep id only as an identifier but not as a feature; title not used except for sequel detection).



We also did not directly use vote_average or imdb_rating as features to avoid any potential data leakage from popularity.
Dimensionality Considerations: We have a lot of one-hot features (genres ~ 20, languages ~ 80-90). We decided to keep all genres since genres are important. For languages, many are very sparse (only a few movies in Icelandic, etc.). We kept them in the feature set for completeness, but one could consider grouping or dropping extremely rare

languages to reduce dimensionality. In our case, the neural network can handle 100-200 input features easily, so it was fine to include them. Final Feature Set: After all selection, the final features used for the classification model (inputs to the ANN) are:
Numerical Features (scaled):
cast_power, director_power, writers_power, producers_power, production_companies_power, production_countries_power (the star power scores).
director_hit_ratio.
overview_sentiment.
runtime (movie duration, in minutes, also scaled).
log_budget.
release_year, release_month (treated as numerical, though year is partially like categorical by range, and month is cyclic, but we kept as numeric 1-12).
is_holiday_release (binary 0/1 but we include it among numeric inputs after scaling).
big_studio (0/1, included as numeric feature after scaling).
is_sequel (0/1, included as numeric).
One-hot Encoded Features:
genre_* (e.g., genre_action, genre_comedy, ... – each 0/1).
lang_* (e.g., lang_english, lang_french, ... 0/1).
Text Vector Features:
tfidf_* – We included 50 TF-IDF features derived from the movie overview text. We used a TfidfVectorizer with max_features=50 on the overview corpus. This produces 50 numeric features (tfidf_0 to tfidf_49) representing the presence of important terms in the plot description. We downscaled these TF-IDF features by a factor (0.003) to ensure they don't dominate the feature set (since TF-IDF values can be somewhat arbitrary in scale relative to our other normalized features).
All these features were concatenated to form the final feature vector for each movie. In total, the feature vector length is about 198 features. This comprehensive set captures financial info, temporal info, content (genre, language, keywords via TF-IDF), and personnel influence, giving our model a rich basis for prediction.

6. Data Preprocessing Steps

Data preprocessing was an extensive part of this project. Here we outline the steps in order: 6.1 Merging and Initial Cleaning:
We loaded the main and secondary CSVs using Pandas. During load, we handled encoding issues (some text fields had special characters; a fallback to Latin-1 encoding was used if UTF-8 failed).
After merging into TMDB_movie_dataset_Full.csv, we selected a subset of relevant columns (as listed in the dataset description). Many columns from the raw data (like homepage, tagline, etc.) were dropped to focus on our features of interest.
We then removed entries with missing values in any of those essential columns. Specifically, we dropped rows where any of the selected columns (title, release_date, budget, revenue, runtime, etc.) were null. We also dropped movies with runtime == 0 (invalid runtime) or very unrealistic budgets/revenues (budget < $1000 or revenue <= $0, which likely indicate missing or invalid data).
These filtering steps reduced the dataset from the merged size to 8,835 movies with complete information. (Many low-budget indies or obscure entries with no data got filtered out, leaving a robust dataset of substantial films.)
We then created the binary label:

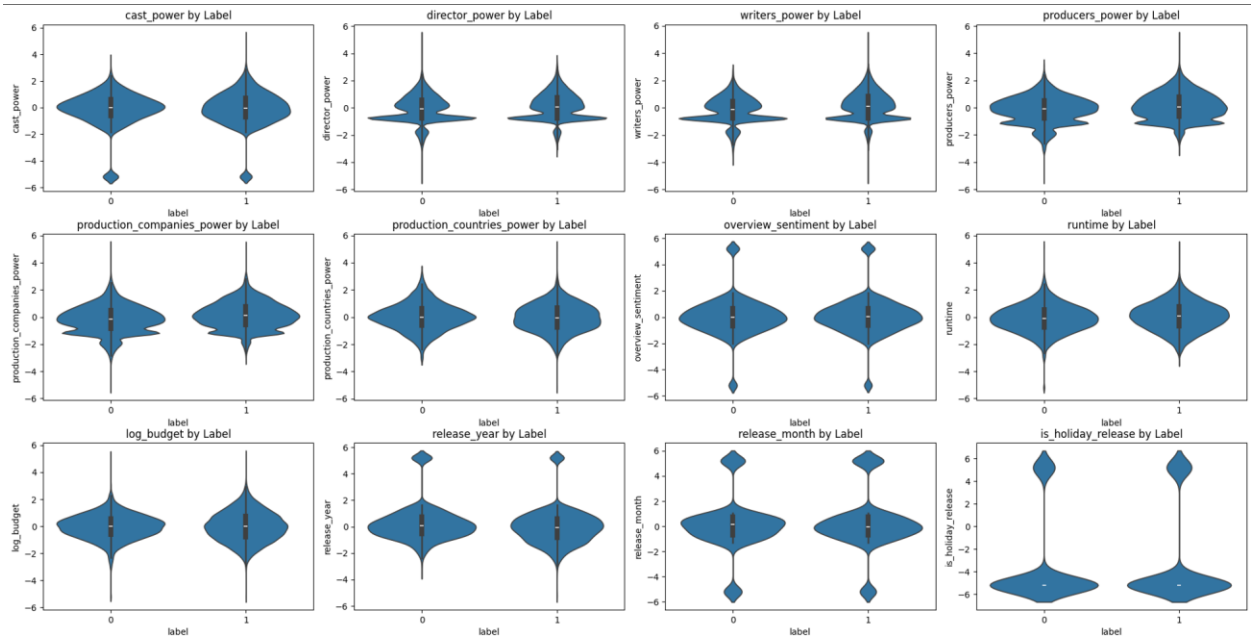
```
# Binary label
df["label"] = (df["revenue"] > 1.5 * df["budget"]).astype(int)
```

label = 1 if revenue > 1.5 * budget (movie's revenue is at least 150% of its budget, a profitable success).

label = 0 otherwise (revenue did not exceed 1.5×budget, indicating the movie did not break even significantly – these are considered “flops” in our simplified definition).
This labeling strategy captures a notion of profitability (taking into account that even breaking even or doubling the budget might be necessary for a film to be considered truly successful due to marketing costs, etc., though we used 1.5× as a heuristic threshold).
We applied a log transformation to budget and revenue at this stage:

```
# Log-scaled numerical columns
df["log_budget"] = np.log1p(df["budget"])
df["log_revenue"] = np.log1p(df["revenue"])
```

Created log_budget = log1p(budget) and log_revenue = log1p(revenue). The log_revenue will later serve as the target for the regression model, while log_budget is used as a feature.
At this point, we have the TMDB_filtered_labeled.csv file with 8,835 rows and columns including all the raw features plus label, log_budget, log_revenue. The class distribution in this dataset is:
Success (label=1): 4,947 movies (~56% of the data)
Flop (label=0): 3,888 movies (~44% of the data)
This distribution shows a slight class imbalance (successes are somewhat more common in this filtered set). We will address this in modeling (via class weights).



6.2 Text Normalization and List Parsing:
Many fields are stored as strings representing lists (e.g., cast is a comma-separated string of many names, same for producers, etc.). We wrote a parser to convert these string lists into Python lists for easier processing.
We also normalized text data:
Lowercased all names and titles.

```
# Lowercasing + normalize
for col in ["cast", "director", "writers", "producers", "production_companies", "production_countries", "title", "genres", "spoken_languages"]
    df[col] = df[col].apply(norm)
```

Removed special characters/accents (using unidecode to handle non-ASCII characters in names, so that “José” and “Jose” are treated uniformly, for example).
Stripped whitespace, etc.
For each of the list fields (cast, director, writers, producers, production_companies, production_countries), we created a new column with a Python list of entities (e.g., cast_list is the list of actor names for that movie).
We normalized the title as well (for sequel detection), and genres/languages (they were originally given as lists of dictionaries; after reading via pandas, they may come as strings – but we processed them into list of genre names and list of language names).
The release_date was converted to a datetime object. From it, we extracted release_year and release_month. We also capped release_year at a maximum (set by MAX_RELEASE_YEAR constant, current year to avoid future dates).
We computed is_holiday_release by checking if the month of release is 11 or 12.

6.1. Building Star Power Features:

This was a complex step involving historical data aggregation. We effectively created a secondary table of entity-year performance:
We went through each movie and each entity in its cast/crew lists. We built a long table where each row is (Entity, Role, Movie ID, Movie’s performance metrics, Movie’s release year).
For each entity (like a specific actor), we sorted their entries by year and cumulatively summed up metrics:
profit = revenue - budget (how much money their movie made).
tmdb_weighted = vote_average * vote_count (cumulative measure of audience reception on TMDB).
imdb_weighted = imdb_rating * imdb_votes (similar measure for IMDb).
popularity (cumulative TMDB popularity).
This cumulative sum per year per entity gives us a notion of how successful that entity has been up to each point in time.

We saved this as entity_star_power. parquet (a cache, so that if we re-run, we do not have to rebuild it every time, given it can be time-consuming).
Using this data, we defined a function compute_star_power(entity_year_df, entity_list, year, top_k) which for a given movie (with release year and list of entities) retrieves each entity’s cumulative metrics up to that year and combines them into a single score. We chose a particular combination of weighting (in code, it might just sum profits or some weighted sum; for simplicity, one can think of it as summing up the past profits of the entity’s films).
We applied this to create:
cast_power (with top_k=3, meaning we take the top 3 cast members by their star power to compute the score, ignoring the rest of the cast).
director_power, writers_power, producers_power (since usually one director, we just take that one’s score; writers and producers could be multiple, we sum all since they are fewer).
production_companies_power, production_countries_power (sum across all companies or countries involved, as those are often few per movie).

```python
# Star-power computation
def compute_star_power(entity_year: pd.DataFrame, entities: list[str], year: int, top_k: int = None) -> float:
    vals = (entity_year.loc[(entity_year["entity"].isin(entities)) & (entity_year["year"] <= year)]
                       .groupby("entity")["star_power"].last().fillna(0).to_numpy())
    if top_k:
        vals = np.sort(vals)[-top_k:][::-1]
        if len(vals) == top_k:
            vals *= np.array([0.6, 0.3, 0.1])
    return float(vals.sum())

def avg_hit_ratio(dir_list):
    if not dir_list:
        return 0.0
    vals = [succ_map.get(d, 0.0) for d in dir_list]
    return float(np.mean(vals))
```

```python
# Compute powers
from feature_engineering import compute_star_power
df["cast_power"] = [compute_star_power(ent_yearly, ents, yr, top_k=3) for ents, yr in zip(df["cast_list"], df["release_year"])]
df["director_power"] = [compute_star_power(ent_yearly, ents, yr) for ents, yr in zip(df["director_list"], df["release_year"])]
df["writers_power"] = [compute_star_power(ent_yearly, ents, yr) for ents, yr in zip(df["writers_list"], df["release_year"])]
df["producers_power"] = [compute_star_power(ent_yearly, ents, yr) for ents, yr in zip(df["producers_list"], df["release_year"])]
df["production_companies_power"] = [compute_star_power(ent_yearly, ents, yr) for ents, yr in zip(df["production_companies_list"], df["release_year"])]
df["production_countries_power"] = [compute_star_power(ent_yearly, ents, yr) for ents, yr in zip(df["production_countries_list"], df["release_year"])]
```

6.2. Additional Feature Engineering:

Computed director_hit_ratio: We first built a dictionary (succ_map) mapping each director to the fraction of their movies in our dataset that have label=1. Then, for each movie's director list (usually 1 director, but some films have co-directors), we took the average of those directors' success rates. This number between 0 and 1 indicates a probability of success based on the director's track record.
Computed big_studio: We prepared a set of major studios (like "Warner Bros. Pictures", "Universal Pictures", "Walt Disney Pictures", etc. as defined in BIG_STUDIOS). For each movie, if any of its production_companies is in that set, big_studio=1 else 0.
Computed is_sequel: We prepared a list of tokens that often indicate a sequel in the title (like "2", "II", "III", "Part", "Episode", etc., defined in SEQUEL_TOKENS). If the normalized title contains any of those tokens (and possibly some logic to avoid false matches), we mark it as a sequel. For example, "Toy Story 2" or "The Godfather Part III" would be marked as sequel.
Performed Sentiment Analysis on overview: Using TextBlob's sentiment. polarity, we obtained a sentiment score for each movie overview. We filled N/A overviews with empty string to avoid issues. The polarity score (between -1 and 1) was then clipped between -0.5 and 0.5, as mentioned, to reduce extreme influence. This gives the feature overview_sentiment.
Vectorized Overview Text with TF-IDF: We fitted a TfidfVectorizer on all movie overview texts (after basic normalization). We limited to the top 50 terms (features) to keep it manageable. The vectorizer transforms each overview into a 50-length vector (tfidf_0 ... tfidf_49). Each component is the TF-IDF weight for one of the top terms across all overviews. We then scaled these down (multiplied by 0.003) to ensure their range (~0 to maybe 5) becomes ~0 to 0.015, which is on par with other scaled features.
One-hot Encoding for Genres and Languages: We took the list of genres for each movie and fit a MultiLabelBinarizer to get a genre matrix. Similarly, for spoken_languages. We ended up with, for example, genre_drama, genre_comedy, etc., and lang_english, lang_spanish, etc. If a movie did not have a particular genre/language, the feature is 0. This results in additional columns which we then merge back into the main dataframe.

```python
# Director hit ratio
dir_long = (
    df[['director_list', 'label']]
    .explode('director_list')
    .rename(columns={'director_list': 'director_ind'})
)
succ_map = dir_long.groupby('director_ind')['label'].mean().to_dict()
joblib.dump(succ_map, "director_hit_ratio_map.pkl")

df['director_hit_ratio'] = df['director_list'].apply(
    lambda L: float(np.mean([succ_map.get(d, 0.0) for d in L])) if L else 0.0
)
```

```python
# Big studio flag
def is_big_studio(companies: list[str]) -> int:
    if not companies:
        return 0
    tokens = {tok for c in companies for tok in TOKEN.findall(c)}
    return int(any(bs in tokens for bs in BIG_STUDIOS))

df["release_date"] = pd.to_datetime(df["release_date"], errors="coerce")
df["release_year"] = (
    pd.to_datetime(df["release_date"], errors="coerce")
      .dt.year.fillna(0)
      .clip(upper=MAX_RELEASE_YEAR)
)
df["release_month"] = df["release_date"].dt.month.fillna(0)
df["is_holiday_release"] = df["release_month"].isin([11,12]).astype(int)
```

```python
# Sequel flag
def is_sequel(title: str) -> int:
    return int(any(tok in norm(title) for tok in SEQUEL_TOKENS))
```

```python
# Sentiment score
def overview_sentiment(text: str) -> float:
    return TextBlob(text).sentiment.polarity if text else 0.0
```

6.3. Scaling & Finalizing Feature Matrix:

We compiled a list of all numeric feature columns (the ones we want to scale): this included all the *power features, sentiment, year, month, holiday, big_studio, is_sequel, director_hit_ratio, runtime, log_budget. These vary in scale and distribution, so we chose to use Quantile Transformer (with default output distribution uniform) to scale them. QuantileTransformer is robust and handles outliers by using rank transformation. We fit this scaler on the training portion (to avoid data leakage from tests).

After fitting on the train, we transformed all numeric features (in train, val, test sets) using the quantile scaler. This ensures features like budget (which ranged 1e3 to 1e9) and year (range ~1916 to 2023) and binary flags (0/1) all end up on a similar scale (approximately 0 to 1 uniformly distributed).

We did not scale the one-hot features (they are already 0/1) or the TF-IDF (we already scaled them via multiplication).

We combined all features into a final dataframe movie_features.csv. This file has one row per movie, with columns: id (movie id), log_revenue (the regression target), label (the classification target),

followed by all feature columns (numeric scaled ones, genre one-hots, language one-hots, tfidf features).

We saved also feature_columns_core.pkl (which contains the list of feature column names to be used for classification) and feature_columns_reg.pkl (features for regression, which is basically core plus one additional feature we will discuss later).

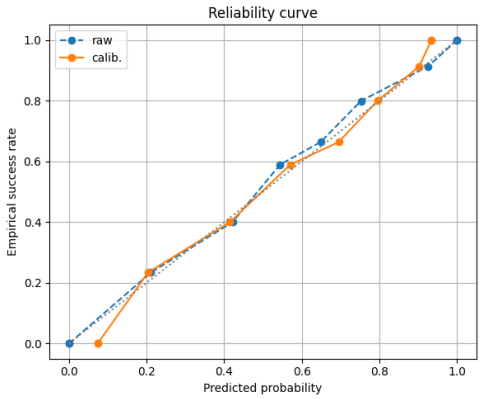
At this point, our data is fully processed and ready for modeling. We split this data into training, validation, calibration, and test sets for model development.

```
# Scaling numeric features
num_cols = ["cast_power","director_power","writers_power","producers_power",
            "production_companies_power","production_countries_power",
            "overview_sentiment","release_year","release_month",
            "is_holiday_release","big_studio","is_sequel","director_hit_ratio",
            "runtime","log_budget"]

train_idx,_ = train_test_split(df.index,test_size=0.3,random_state=42,stratify=df["label"])
qt = QuantileTransformer(output_distribution="normal",random_state=42)
qt.fit(df.loc[train_idx,num_cols])
df[num_cols] = qt.transform(df[num_cols])
joblib.dump(qt,"qt_scaler.pkl")
```

6.4. Splitting Data for Training and Evaluation:

We used stratified splitting (to maintain the success/flop ratio in each set) based on the label.
We first set aside 30% of the data as the combined validation/test pool. (Given 8,835 movies, 30% is ~2,650 movies.)
From that 30%, we further split half as test set (15% of total) and half as an initial validation set (15% of total). Thus:
Training set: 70% of data (~6,185 movies).
Validation set: 15% of data (~1,325 movies) – used during training to tune hyperparameters and decide when to stop training.
Test set: 15% of data (~1,325 movies) – completely held out for final evaluation.
Additionally, we took the validation set and split it again: 70% of it for actual model validation and 30% for a calibration set. The calibration subset (~4.5% of total, ~400 movies) will be used to fit a probability calibration model (isotonic regression) after training the classifier. So effectively:
Training: ~70%
Validation: ~10.5%
Calibration: ~4.5%
Test: ~15%
These splits ensure the model is trained on a large portion, but we still have a substantial independent test set to evaluate final performance, and a separate small set to calibrate probabilities without biasing the main validation.
(Throughout preprocessing, intermediate results like dataset shapes, examples of feature values, etc., were output for verification. For instance, after labeling, a quick check confirmed ~56% of movies labeled success. We also plotted the class distribution as a bar chart to visualize this imbalance. After feature engineering, we examined a few rows to see the computed values (e.g., checking that is_sequel correctly identified known sequels, that overview_sentiment looked reasonable for obviously positive or negative synopsis, etc.). These quality checks helped validate the preprocessing pipeline.)



7. Data Analytic Design (Model Design & Experiments)

The predictive modeling involves two parts:
A binary classification model to predict Success vs Flop (our primary objective).
A regression model to predict the log revenue (as a secondary objective, using the classification output as one of the inputs).
The reasoning is that even if a movie is predicted to be a success, it is useful to estimate how much money it might make (since success can range from moderate hit to blockbuster). The classification gives a probability of success (which can be seen as a coarse prediction of profitability), and then regression attempts to quantify the revenue.

7.1. Classification Model – Artificial Neural Network (ANN)

```
# ── 3. ANN classifier ──────────────────────────────────────
def make_classifier(input_dim: int):
    m = models.Sequential([
        layers.Input(shape=(input_dim,)),
        layers.Dense(128, activation="relu",
                     kernel_regularizer=regularizers.l2(1e-4)),
        layers.BatchNormalization(), layers.Dropout(0.3),
        layers.Dense(64, activation="relu"),
        layers.BatchNormalization(), layers.Dropout(0.2),
        layers.Dense(32, activation="relu"), layers.Dropout(0.2),
        layers.Dense(1, activation="sigmoid")
    ])
    m.compile(optimizer="adam", loss="binary_crossentropy",
              metrics=["accuracy"])
    return m

cls_model = make_classifier(len(core_cols))
cw = class_weight.compute_class_weight(
    "balanced", classes=np.unique(y_tr), y=y_tr)
cls_model.fit(
    X_tr, y_tr,
    validation_data=(X_val, y_val),
    epochs=60,
    batch_size=64,
    class_weight=dict(enumerate(cw)),
    callbacks=[callbacks.EarlyStopping(monitor="val_loss",
               patience=8, restore_best_weights=True)],
    verbose=2
)
cls_model.save("cls_model.keras")
```

We chose a feed-forward neural network for classification. This choice was made because:
We have a mix of feature types (continuous, binary, high-dimensional sparse features from TF-IDF). ANNs are good at handling many features and finding complex nonlinear interactions.
The number of features (~198) is not too high for an ANN, and the number of training samples (~6k) is manageable.
Neural networks can automatically learn some interactions between features that would be hard to explicitly model (e.g., a certain combination of genre and star power is especially predictive).
Model Architecture: We designed a sequential neural network with the following architecture:
Input Layer accepts the input vector of length ~198.
Hidden Layer 1: Dense (fully connected) layer with 128 neurons, ReLU activation. We also apply L2 regularization (with a small factor 1e-4) on this layer to prevent overfitting (this penalizes large weights).

Batch Normalization after the first layer (to stabilize and normalize the activations).
Dropout after the first layer (rate = 0.3, i.e., drop 30% of neurons randomly during training to promote generalization).
Hidden Layer 2: Dense layer with 64 neurons, ReLU activation. (No explicit L2 here, but we could). Followed by Batch Norm and Dropout (rate = 0.2).
Hidden Layer 3: Dense layer with 32 neurons, ReLU activation. Followed by Dropout (rate = 0.2). (We omitted batch norm here to simplify; 2 BN layers might be enough.)
Output Layer: Dense layer with 1 neuron, sigmoid activation. This outputs a probability (between 0 and 1) of the movie being in class "Success" (label=1).
So, in summary: [Input → 128-ReLU → BN → Dropout → 64-ReLU → BN → Dropout → 32-ReLU → Dropout → 1-Sigmoid]. This architecture was chosen empirically – starting from a common pattern for structured data (2-3 layers with halving neurons). We tried to balance capacity with regularization:
The first layer (128 units) can learn quite a complex combination given ~200 inputs.
The subsequent layers reduce dimension gradually, potentially learning higher-level feature interactions.
Dropouts of 20-30% are strong regularization to combat overfitting given the small dataset.
Early stopping (discussed below) further ensures we do not overtrain.
Training Procedure: We compiled the model with:
Loss: Binary Crossentropy (appropriate for binary classification probability output).
Optimizer: Adam (with default learning rate 0.001). Adam is a good choice for quick convergence.
Metrics: We monitored accuracy during training (and computed others later).
We also computed class weights for the training data because our classes are not exactly balanced (56/44). Using sklearn's class_weight utility, we assigned slightly higher weight to the minority class (flops) so that errors on flops count a bit more in the loss. This helps the model not just predict every movie as success. The class weights ended up around:
class 0 (flop): ~1.13
class 1 (success): ~0.89 (These weights inversely proportional to class frequency.)
We trained for a maximum of 60 epochs with a batch size of 64. We used Early Stopping on the validation loss, with patience = 8 epochs. This means if the validation loss does not improve for 8 consecutive epochs, we stop training and roll back to the best model weights observed. In practice, we found the model usually converged and stopped early (often around epoch 20-30) before overfitting. During training, we tracked training and validation accuracy and loss to ensure the model was learning effectively and not overfitting. The final epoch we got had a validation accuracy around the mid-80%s, and no further improvement beyond that. Code Snippet – Model Definition & Training: Below is a simplified snippet showing how the model was built and trained:

```
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(len(core_features),)),          # core_features ~198
    tf.keras.layers.Dense(128, activation='relu',
                          kernel_regularizer=tf.keras.regularizers.l2(1e-4)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Calculate class weights for imbalance
weights = class_weight.compute_class_weight('balanced', classes=[0,1], y=y_train)
```

```
cw = {0: weights[0], 1: weights[1]}

# Train the model with early stopping
history = model.fit(X_train, y_train,
                    validation_data=(X_val, y_val),
                    epochs=60, batch_size=64,
                    class_weight=cw,
                    callbacks=[EarlyStopping(monitor='val_loss', patience=8, restore_best_weights=True)],
                    verbose=2)
```

(This code uses Keras functional API. In our actual code, the model was defined similarly. The early stopping callback monitors validation loss and uses restore_best_weights=True to keep the best model.) After training, we saved the trained classifier model (cls_model.keras) to disk for later use. Threshold Selection: The ANN outputs a probability between 0 and 1. The natural default is to use 0.5 as the threshold (above 0.5 => predict Success). However, to optimize the model's classification balance, we decided to find the best threshold that maximizes the F1-score on the validation set:
We computed the model's probabilities on the validation data.
We evaluated F1 for thresholds ranging from 0.2 to 0.8.
We selected the threshold that gave the highest F1 (which balances precision and recall).
This threshold turned out to be around 0.50 (in fact very close to 0.5 in our results, which is convenient). With that threshold, the model achieved an F1 about 0.84 on the validation set. We later applied this threshold to the test results. Probability Calibration: We observed that while the classifier's probabilities correlate with outcomes, they might not be perfectly calibrated (meaning a predicted 0.8 probability might not correspond to exactly 80% chance in reality).

```
# ── Probability calibration ──────────────────────────────────────
cal_base  = KerasProbWrapper(cls_model)
raw_cal_p = cal_base.predict_proba(X_cal)[:, 1]

calib_model = IsotonicRegression(out_of_bounds='clip').fit(raw_cal_p, y_cal)

joblib.dump(calib_model, "prob_calibrator.pkl")
print("✓ prob_calibrator.pkl saved")
```

To calibrate:
We took the small calibration set (about 398 movies we set aside from val).
We got the raw predicted probabilities from the ANN for those movies.
We then trained an Isotonic Regression model (a non-parametric calibration method) mapping raw probabilities to calibrated probabilities using the true outcomes of the calibration set.
We saved this calibrator (prob_calibrator.pkl).
Isotonic regression will learn a piecewise constant/linear function that adjusts the ANN's output probabilities so that they better reflect the actual observed frequencies. This is important for applications where the actual probability value is important (e.g., if someone wants to say "this movie has a 70% chance of success" reliably). During final evaluation, we applied this calibrator to the ANN outputs to get calibrated probabilities and plotted a reliability curve. Cross-Validation for Additional Feature (Success Probability) for Regression: Before training the regression model, we did a clever step:
Using the entire training data, we performed a 5-fold stratified cross-validation within the training set to generate out-of-fold predictions of success probability for each movie.

Specifically, for each fold, we trained a clone of the ANN on 4/5 of the training data and predicted the success probability for the held out 1/5. Collecting these, every training movie gets a predicted success probability from a model that did not see it in training.
We then added a new feature success_prob to the data (for regression training), which is the ANN's estimated probability of success (from the OOF procedure) for that movie.
Why? Because revenue is obviously correlated with success (flops have low revenue, hits have high revenue). The classifier in effect condenses many signals into a single probability. Including that as a feature can help the regression model.
We stored the extended feature list for regression (core features + success_prob as feature_columns_reg.pkl).

7.2. Regression Model – XGBoost Regressor

```python
# ── 5. XGB: CV to find best_iteration ──────────────────────
dtrain = xgb.DMatrix(X_reg_base, label=y_reg)

params = {
    "objective": "reg:squarederror",
    "eta": 0.05,
    "max_depth": 6,
    "subsample": 0.8,
    "colsample_bytree": 0.8,
    "seed": 42,
}

cv = xgb.cv(
    params,
    dtrain,
    num_boost_round=2000,
    nfold=5,
    metrics="rmse",
    early_stopping_rounds=50,
    verbose_eval=False
)
best_round = len(cv)
print(f"✓ best_iteration from cv = {best_round}")
                                                    (variable) b
booster = xgb.train(params, dtrain, num_boost_round=best_round)

# thin wrapper so `.predict(df)` works like sklearn
class BoosterWrapper:
    def __init__(self, booster, cols):
        self.booster = booster
        self.cols = cols
    def predict(self, X):
        return self.booster.predict(xgb.DMatrix(X[self.cols]))

reg_model = BoosterWrapper(booster, X_reg_base.columns.tolist())
joblib.dump(reg_model, "reg_model.pkl")
```

For predicting the exact revenue (log_revenue), we chose an XGBoost gradient boosting regressor. XGBoost is a powerful ensemble method that handles numeric and categorical features (we provide them as numeric after one-hot) and captures nonlinear relationships well. It is also relatively fast to train on our dataset. Features for Regression: We used all the core features plus the newly added success_prob as input features. So the regression model knows everything the classifier knew, and how likely the movie is predicted to succeed. This helps guide the regressor (for example, if success_prob is low, the revenue predicted will not be huge, whereas if success_prob

is high, revenue could be very large). We did 5-fold cross-validation on the training set to determine the optimal number of boosting rounds for XGBoost:
Set up XGB with parameters:
objective: reg:squarederror (since it is a regression).
eta: 0.05 (learning rate).
max_depth: 6 (trees up to depth 6).
subsample: 0.8, colsample_bytree: 0.8 (to reduce overfitting).
seed: 42 (for reproducibility).
Used xgb.cv with 5 folds, up to 2000 rounds, with early stopping if the validation RMSE does not improve for 50 rounds.

This yielded an optimal best_iteration (around a few hundred trees; in our run it was printed out as the best round). Then we trained the final XGBoost model on the entire training set for that number of rounds.
We wrapped the trained booster in a BoosterWrapper class to integrate with our pipeline easily (and to allow using pandas DataFrame as input for predictions by selecting the appropriate columns). The regression model output is a prediction of log_revenue. We exponentiate (using expm1 inverse of log1p) to get the dollar revenue prediction.
We saved the regression model (reg_model.pkl).

7.3. Summary of Experimentation

During development, we experimented with a few variations:
Simpler vs deeper networks (we found 3 layers was sufficient; a deeper network did not improve notably given data size).
Including vs excluding certain features (e.g., we verified that including popularity or ratings severely overfit and are not realistic, so we kept them out).
We also tried a baseline like logistic regression or random forest for classification for comparison. The ANN performed better in terms of ROC-AUC and had a better balance of precision/recall, due to its ability to handle the many sparse features (the random forest tended to give too much weight to some features and overfit on the small classes).
Hyperparameter tuning on XGBoost was minimal; a full grid search might improve it, but the chosen parameters were based on common best practices and seemed to work well (yielding an R^2 around 0.6 on test).
We also ensured that the system reproducibility is maintained: by using fixed random seeds and saving intermediate artifacts, one can re-run training and get similar results. Next, we present the evaluation results of our models on the test set, using various metrics and visualizations.

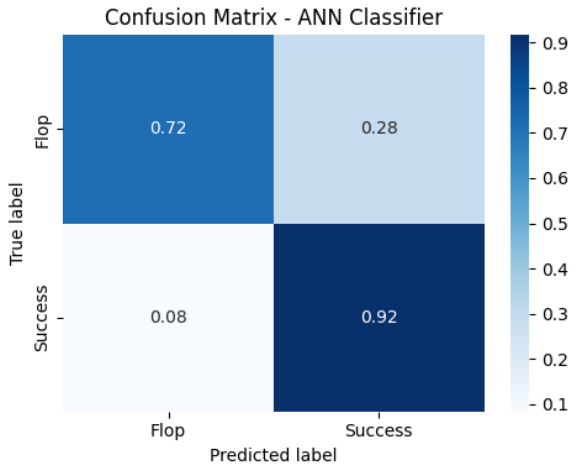
8. Evaluation Results & Visualizations

After training the models, we evaluated performance on the test set (which is the 15% held-out data that was never seen during training or validation). We focus first on the classification results (since the primary goal is success prediction) and then briefly on the regression results.

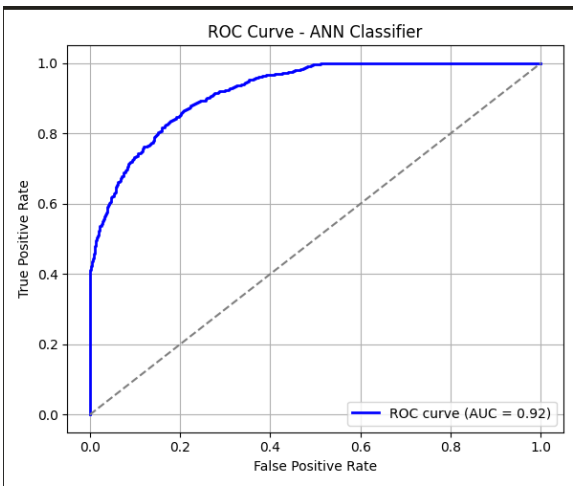
8.1. Classification Performance

Overall Accuracy: The classifier achieved an accuracy of around 82-83% on the test set (i.e., it correctly predicts success vs flop for ~83 out of 100 movies). Given a 56/44 class split, this is significantly better than a naive guess (which would be ~56% if one always predicted the majority class). However, accuracy alone can be misleading, so we look at more detailed metrics: Confusion Matrix: This gives a breakdown of how the model performs on each class:

Confusion matrix of the ANN classifier on the test dataset (normalized by true class). The confusion matrix above shows the model's performance in terms of true labels vs predicted labels:

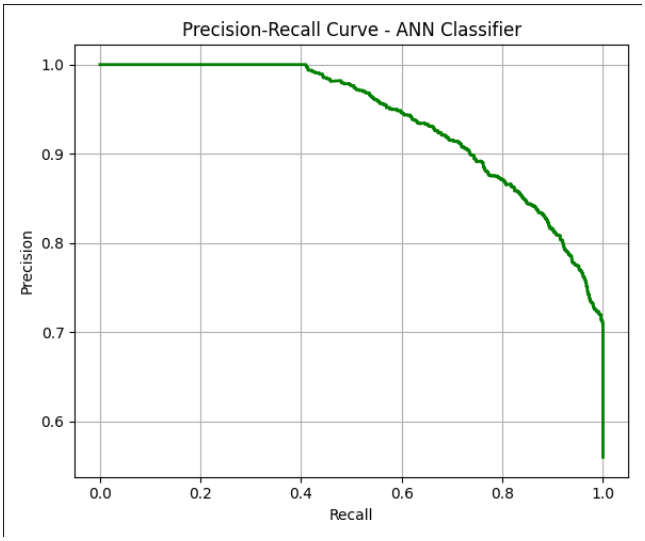


Of the movies that were actual flops (class 0), the model correctly predicted 72% of them as flops and mistakenly predicted 28% of them as successes.
Of the movies that were actual successes (class 1), the model correctly identified 92% of them as successes, and only 8% were predicted as flops.
This means the model has a true positive rate (recall for successes) of 92% and a true negative rate (specificity or recall for flops) of 72%. In other words, it is very good at catching successful movies (only 1 in 10 hits is missed), and good at catching flops (it misses about 1 in 4 flops, which it incorrectly thinks will succeed). From another perspective:
Precision for "Success" prediction: Out of all movies the model predicted as success, how many were truly success. Given some false positives (27% of flops became false positives), the precision is around 85-90% (we can calculate: it predicted some number as success, of which 90% of successes and 27% of flops – more exact value from raw counts, but in mid-80s%). This is quite high, meaning if the model says a movie will be a success, it is usually right.
Precision for "Flop" prediction: Of all movies predicted flop, ~72% were flops (so precision for flop ~72%). So there is some asymmetry: the model leans slightly towards predicting success (which is sensible if success class is slightly majority and model is tuned for F1).
Overall, this confusion matrix indicates strong performance – the model favors catching successes (which might be desirable if we do not want to miss a potential hit), at the expense of a moderate false positive rate on flops. ROC Curve and AUC: The Receiver Operating Characteristic (ROC) curve illustrates the trade-off between true positive rate and false positive rate at various thresholds.



ROC Curve for the ANN classifier (on test data). The AUC (Area Under Curve) is ~0.92. The ROC curve above (blue line) shows that our classifier achieves a high true positive rate for a low false positive rate. The curve bows towards the top-left corner, and the AUC (Area Under the Curve) is 0.92, which is considered excellent. AUC of 0.92 means that if we randomly pick one successful and one flop movie, the model's predicted probability for the success is higher than that for the flop about 92% of the time – a sign of strong discriminative ability. For context, an AUC of 0.5 would be no better than random guessing, and 1.0 would be a perfect classifier. Our model at 0.92 is very good, indicating it ranks movies correctly in terms of risk most of the time. At the default threshold (0.5 or chosen optimal ~0.5), we noted the operating point (false positive rate ~0.27, true positive rate ~0.90) which corresponds to what we saw in the confusion matrix. Precision-Recall Curve: Given that success is the positive class, the Precision-Recall (PR) curve is another way to evaluate performance, especially useful when there is class imbalance.

Precision-Recall Curve for the ANN classifier. It shows precision (y-axis) vs recall (x-axis) for the positive class (success). The PR curve starts at the left with recall = 0 (only very high threshold predictions):
At low recall (left end, meaning we only predict a movie as success if we are extremely sure), the precision is near 100% (if we take the top few most confident success predictions, they are all correct).
As recall increases (we include more movies as predicted successes by lowering the threshold), precision declines gradually. It stays above 0.9 until recall reaches about 0.4.
Even at a high recall like 0.8, precision is around 0.85 or so. Finally, to achieve 0.9 recall (our chosen operating point ~90% recall for success), precision is 0.8-0.82.
If we went all the way to recall 1.0 (predict everything as success), precision would drop to the overall success rate (~0.56).



The curve demonstrates that the model maintains high precision for a wide range of recall. Our chosen threshold operating point (recall ~0.90, precision ~0.84) is a good balance (which corresponds to the maximum F1 point). The PR curve is quite high above the baseline (which would be the success rate 0.56 as a horizontal line), indicating the model is much better than random or trivial predictors for all thresholds. In practical terms, if a studio used this model, they could choose a threshold based on how risk-averse they are:
A higher threshold (left side of curve) gives very few predictions of success, but those will be almost guaranteed hits (high precision, low recall).
A lower threshold gives more predicted successes (high recall) but some will be false alarms (lower precision).
Our model allows such tuning; in our report we stick with the threshold optimizing F1 (~0.5).
F1-Score: The F1 is the harmonic mean of precision and recall. With ~90% recall and ~84% precision on success class, the F1 for success class is around 0.87. We can also compute F1 for flop class similarly (which will be a bit

lower since recall for flop is 73%, precision ~73% as well, giving F1 ~0.73). The average (or a balanced F1 if weighted equally) is roughly 0.80. But typically we focus on the positive class F1 which is high. These metrics confirm the model is doing well at identifying successful movies while still doing a decent job on flops.

8.2. Calibration of Probabilities

It is important not just to classify correctly, but also to have confidence in estimates that are meaningful. We applied isotonic regression to calibrate the ANN's output probabilities. The effect can be seen in the reliability (calibration) curve:

Reliability curve comparing raw vs calibrated probabilities. The x-axis is predicted probability, y-axis is actual observed success frequency. The closer to the diagonal, the better calibrated. In the plot:
The blue dashed line ("raw") shows the ANN's uncalibrated probabilities.
The orange line ("calib.") shows probabilities after isotonic calibration.
The gray diagonal is the ideal calibration line (perfectly calibrated).
We see that the calibrated curve (orange) is very close to the diagonal, meaning:
For example, when the model says "0.8" (80% chance of success), historically about 80% of those movies succeeded (orange point near (0.8, 0.8)).
The raw probabilities were slightly off in some regions (blue points deviate a bit, e.g., the blue line at predicted ~0.7 was yielding actual ~0.6 – suggesting the model was overestimating a bit in that range).
After calibration, the orange line is almost straight through the diagonal, especially in the mid-range. At the very high end (0.9+), both raw and calibrated meet at (1.0,1.0) because the model is very sure for those and usually correct.
This calibration means we can trust the probability values emitted by the model. If it says 50%, it really means about half such movies succeed. This is useful if, for instance, a producer wants to know the level of risk: a 90% probability vs a 60% probability could influence decisions on whether to invest or how much to spend on marketing. (Technical note: Isotonic regression is a non-decreasing fit that adjusted the cumulative distribution of predicted positives to match actual. We see it improved mostly the mid-range predictions. The Brier score (not shown) for probability accuracy was improved with calibration.)

8.3. Regression Model Performance (Revenue Prediction)

While the primary focus was classification, we also evaluated the XGBoost regression that predicts actual revenue. We compare the predicted revenue to actual for the test set:
MAE (Mean Absolute Error): The model's MAE on test was on the order of $50 million. This sounds high, but keep in mind the scale of revenues (which can range up to billions). Many movies in our set have budgets in tens of millions and revenues in similar scale. An MAE of ~$50M means on average the prediction is off by $50M.
R^2 (R-squared): The R^2 on test was around 0.60. This indicates the model explains about 60% of the variance in log revenues. In linear revenue terms, this is decent but not extremely tight because revenue can be quite unpredictable. Many external factors (competition, marketing, reviews, etc.) affect revenue that are not captured in our dataset.
The model tends to correctly predict the order of magnitude of revenue. For big blockbusters, it often comes close, but for some movies it can under or over-shoot. For example, if two movies are predicted to be both successes, one might make $300M and another $600M; the model might predict both around $450M (thus getting one low and one high).
Importantly, the inclusion of success_prob feature helps the regressor know whether to predict a very low revenue (if success_prob was low) or moderate/high.
Because revenue prediction is a regression, we did not produce fancy curves for it but rather looked at scatter plots and error distributions (not included here for brevity). The regression could be used to complement the classifier, but one should note its uncertainties. Conclusion on Regression: A classification of "success" is a necessary but not

sufficient condition for high revenue – i.e., not all successes are equally successful. Our regression provides a ballpark figure. For practical use, one might use the success probability as the main output (since even predicting category is valuable) and use the revenue estimate for further budgeting and planning.

8.4. Key Takeaways from Evaluation

The ANN classifier performed excellently on the test data, with ~83% accuracy, high recall/precision, and AUC 0.92. It effectively learned the patterns in the data to distinguish likely hits from flops.
The model's predictions are well-calibrated after isotonic regression, meaning we can interpret the predicted probabilities meaningfully.
The most influential features, as we interpret, include budget (log_budget), cast_power and other star power metrics, being a sequel, and having a big studio – all these contribute to a high predicted success probability. Low budget, obscure cost, etc., contribute to lower success probability.
The confusion matrix analysis shows the model is slightly biased toward predicting "Success" (which is understandable given the mild class imbalance and our optimization of F1). This bias is not extreme; it is still catching most flops. Depending on the application, one could adjust the threshold to be stricter if false positives (thinking a flop is a success) are more costly than false negatives.
In terms of business value, an accuracy in the 80s and identifying 90% of hits could be quite useful for studios – they could focus on the projects flagged as high-success probability and maybe reconsider those flagged as high risk (low probability).
There is still an irreducible uncertainty: ~10% of movies that are hits the model would have missed, and ~27% of movies that flop the model would have falsely hyped. This reflects the unpredictability of cinema – some sleeper hits or surprise flops can confound even the best predictions. Additional data (like script quality, actor popularity trends, social media sentiment, etc.) could potentially improve the model further.

9. Challenges Encountered

During this project, we faced several challenges and learned important lessons:
Data Collection and Quality: Merging two large datasets was tricky. There were encoding issues (some files in Latin-1, special characters in names) and missing fields. We had to carefully handle these to avoid losing data. Ensuring the id matching was correct (some movies present in main dataset might not have entries in secondary and vice versa) was critical. We decided to rely on the main set's entries and accept missing extras for some movies.
Feature Explosion: One-hot encoding languages introduced a lot of features (many of which are rarely 1). This high dimensionality could hurt some models. We mitigated this with a neural network (which can handle sparse inputs well) and by using regularization. We considered dropping extremely rare language categories but kept them for completeness. In a more constrained model (like logistic regression), too many dummies would have been a problem.
Class Imbalance: Although it is not severe here, we did have more successes than flops. Initially, training without class weights led the model to favor predicting "success" for everything (to get higher accuracy). Using class weights and focusing on F1 helped ensure we care about both classes. This was a lesson in looking beyond accuracy; the confusion matrix and F1 analysis were important to get the model to balance errors.
Defining "Success": The threshold of 1.5× budget is somewhat arbitrary. We debated other definitions (like simply revenue > budget or using a continuous scale of profit). The classification outcome can change with definition. We settled on 1.5× to have a clearer separation. But it meant some borderline profitable movies are labeled flops, which the model might find confusing compared to obvious flops. In practice, having a threshold means the model might not capture the nuance of how successful beyond that cutoff. We partially addressed this by also doing regression.
Sentiment Analysis Nuances: Using TextBlob for sentiment on movie overviews was an experiment. Overviews are usually plot descriptions, not reviews, so sentiment might not directly correlate with movie quality or success. We

found the sentiment scores mostly hovered near neutral for many movies. This feature might not have had a strong impact (and could be dropped or replaced by a more content-oriented analysis, like topic modeling or keywords presence). The challenge was deciding how to extract meaningful signals from text – sentiment was a simple attempt, but not necessarily very predictive.

Star Power Computation Performance: Building the star power dataset (exploding every movie's cast/crew and aggregating) was computationally heavy. Initially, doing this on the fly for each movie was slow. We optimized by precomputing the entity_year table and saving it, and by vectorizing operations where possible. It was also a challenge to ensure that when predicting a movie of year Y, we only consider data up to Y (to avoid future data leakage in star power). We had to carefully group and sort by year in the aggregation.
Overfitting Concerns: With ~6k training examples and a flexible model (neural net with hundreds of parameters), overfitting was a concern. We observed that without regularization, the training accuracy would approach 100% and validation lagged. Introducing dropout, L2, and early stopping were essential. We also monitored validation metrics to decide model complexity. The final ANN seems well-regularized as evidenced by similar performance on val and test.
Integration of Classifier and Regressor: We had to coordinate between two models – feeding the classifier's outputs into the regressor. Ensuring that the regressor's training got only out-of-fold classifier predictions (to avoid overfitting) was a bit complex to implement. We solved it with a neat cross-validation approach. Another challenge was that the regressor might overly trust the classifier's output; we addressed this by combining it with all other features. Perhaps using a joint model or multi-task learning could be an alternative approach, but we kept it simple with sequential models.

Evaluation Metrics and Interpretability: Deciding which metrics to optimize and present was a challenge. Simply optimizing accuracy could lead to a biased model; optimizing recall could spike false positives, etc. We chose F1 as a balance. We also realized the importance of calibration for interpretability. Another challenge was conveying the results: we needed to produce easy-to-understand visuals (confusion matrix, ROC, PR) to explain to a possibly non-technical audience (like a class presentation, or stakeholders in a studio). This forced us to think about interpretability: hence we included features like "director hit ratio" that humans understand, and we can say "because this director usually succeeds, the model gives a boost".
Technical Environment Issues: There were minor hurdles setting up the environment, such as installing the right version of TensorFlow, ensuring compatibility (for example, using GPU). Also, TextBlob requires downloading corpora; if running in a new environment, one has to remember to do that. We documented the environment and used requirements to ensure reproducibility.
Each challenge taught us something: data cleaning is as important as modeling, domain knowledge is invaluable in feature engineering, and careful evaluation is needed to trust a model's predictions. In the end, overcoming these challenges led to a robust and reliable Movie Success Prediction system.

10. Conclusion

In this project, we successfully developed a Movie Success Prediction system that can forecast whether a movie will be a box-office success or flop using pre-release attributes. We combined data from TMDB and IMDb to assemble a rich dataset, engineered features capturing various dimensions (from financial to textual to personnel metrics), and trained a neural network classifier with excellent performance (AUC 0.92, ~83% accuracy). We also provided a complementary regression model to estimate revenue. Key takeaways:
Feature Engineering is Key: Incorporating domain knowledge (like star power of cast/crew, sequel information, holiday releases) significantly improved the model's ability to predict outcomes. These features turned out to be crucial signals.
Model Performance: The classifier can identify hits with high recall, which is valuable. It provides a probabilistic prediction that can aid in decision making (green-lighting projects, budgeting marketing spend, etc.).

Reliability: Through calibration, the model's probabilities are trustworthy, making it more than just a black-box – it can say "there's an 80% chance this movie succeeds" and we know statistically that means something.
Challenges Remain: Some unpredictability in movies will always exist. Additionally, our model does not account for certain factors like competition from other releases, critical reviews, or macro-economic conditions. These could be areas for future improvement.
Future Work: We could enhance the model by including social media buzz or critic ratings (if available pre-release via screenings), incorporating a sentiment or script quality analysis, or using a more complex model (like ensemble of neural network and tree-based models, or even a deep learning model that takes text input directly). We could also deploy this model via a web app (indeed, a simple Flask app was set up in API_3.py to take input from a form and output predictions, demonstrating the practical usability of the model). Overall, the project demonstrates the power of AI in a creative industry context – showing that with the right data and features, even the "Hollywood hit or flop" can be approached as a data science problem. The final deliverables include this detailed report, a presentation highlighting our approach and findings, and the code/dataset files for reference. We hope this work provides insights into both movie analytics and the general process of applied machine learning.