Movie Success Predictor

A Comprehensive Analysis and Implementation

DATA 200 - Applied Statistical Analysis

Team Utopia

Prashant Koirala (Project Leader)
Aaska Koirala
Aishmita Yonzan

June 22, 2025

A Comprehensive Technical Documentation

Abstract

Abstract

This documentation provides a comprehensive explanation of the Movie Success Predictor project, which combines data analysis, machine learning, and web development to create an interactive application that predicts whether a movie will be a "hit" or a "flop." The system features a React-based frontend that presents users with movie information from The Movie Database (TMDB) API and a Flask backend that handles data processing and predictions using a logistic regression model. The project demonstrates the practical application of statistical methods and software engineering practices in creating an end-to-end solution for movie success prediction.

Final Project Submission
Applied Statistical Analysis (DATA 200)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Overview

The Movie Success Predictor is an innovative, interactive web application that transforms complex machine learning predictions into an engaging quiz-style game. It allows users to make educated guesses about whether a movie will be commercially successful (a "Hit") or unsuccessful (a "Flop") based on movie information and promotional poster. The system then compares the user's intuition against both a trained machine learning model's prediction and the actual historical outcome.

This multidisciplinary project represents the intersection of several advanced technical domains:

- **Data Analytics:** Processing and analyzing comprehensive movie industry data to identify hidden patterns, correlations, and predictive factors that contribute to box office success. This includes exploring relationships between budget, runtime, genre, audience ratings, and financial performance.
- **Machine Learning:** Training and optimizing a sophisticated logistic regression model that can accurately classify movies as hits or flops based on multiple features. The model incorporates both numerical variables (budget, runtime, ratings) and categorical variables (genre, country of origin, certification) using one-hot encoding techniques.
- **Web Development:** Creating a responsive, modern frontend experience using React and TypeScript that provides an intuitive and engaging interface. The application features interactive components, real-time feedback, animations, and a game-like progression system.
- **API Integration:** Establishing robust connections with The Movie Database (TMDB) for accessing rich movie information, high-quality poster images, and metadata. This integration ensures the application presents users with accurate and visually compelling content.
- **Full-Stack Implementation:** Combining frontend and backend technologies into a cohesive architecture with clear separation of concerns, efficient data flow, and optimized communication protocols.

The application provides several distinct game modes with different difficulty levels, allowing users to challenge themselves while learning about the factors that influence movie financial success. Interactive elements like animations, sound effects, and visual feedback enhance the user experience and maintain engagement throughout the quiz progression.

1.2 Problem Statement and Motivation

The global entertainment industry invests over $100 billion annually in film production and marketing, yet predicting commercial success remains one of the most challenging and elusive goals for studios, producers, and investors. Despite massive budgets and sophisticated marketing campaigns, approximately 70% of movies fail to break even at the box office, resulting in significant financial losses and industry volatility.

1.2.1 Industry Challenges

Movie success prediction presents unique challenges not found in many other prediction domains:

- **High Variance:** A film's success depends on numerous interconnected factors including star power, release timing, competition, cultural trends, and unpredictable audience reception.
- **Subjective Evaluation:** Unlike many products, a film's quality is largely subjective and can be interpreted differently across cultural contexts, demographics, and time periods.
- **Limited Training Data:** While thousands of movies are produced annually, only a fraction provides complete and reliable data for training models.
- **Novelty Factor:** Each film is inherently unique, making direct comparisons challenging even within the same genre or with similar characteristics.
- **External Factors:** Unpredictable events like pandemic restrictions, competing entertainment options, or cultural shifts can dramatically impact success regardless of a film's intrinsic qualities.

1.2.2 Project Objectives

This project addresses these challenges through a data-driven, interactive approach with the following key objectives:

1. **Comprehensive Analysis:** Conduct detailed statistical analysis of historical movie data to identify and quantify the factors that most strongly correlate with commercial success, including production budget, runtime, genre, country of origin, critical reception, and certification rating.
2. **Predictive Modeling:** Develop, train, and rigorously evaluate a machine learning model capable of accurately classifying movies as "Hits" or "Flops" based on objectively available pre-release information, achieving significantly better than random performance.

3. **Educational Interface:** Create an engaging, intuitive interface that allows users to test their own predictive abilities against the machine learning model, learn about significant factors in movie success, and gain insights into the complexity of entertainment industry economics.
4. **Technical Integration:** Demonstrate effective integration of modern web development frameworks with machine learning technologies in a coherent end-to-end application architecture.
5. **Empirical Validation:** Validate both human intuition and machine learning approaches to success prediction through direct comparison on a diverse dataset of real-world movie examples.

1.2.3 Research Questions

The project seeks to address several fundamental research questions:

- Which factors are most predictive of a film's commercial success?
- How does human intuition compare to machine learning approaches in predicting movie success?
- Can an objective, feature-based model outperform random guessing despite the subjective nature of film appreciation?
- What combination of features provides the optimal balance of predictive power and model simplicity?
- How can complex statistical predictions be presented in an engaging, accessible format for non-technical users?

1.3 Target Audience and User Personas

The Movie Success Predictor has been designed with several distinct user groups in mind, each with different needs, technical backgrounds, and engagement goals. Understanding these target audiences informed key design decisions throughout the development process.

1.3.1 Primary Target Audiences

- **Film Enthusiasts and Cinephiles:** Users with extensive movie knowledge who enjoy testing their understanding of what makes movies commercially successful. These users appreciate the depth of movie information, historical context, and the challenge of outperforming the AI model.
- **Data Science Students and Practitioners:** Individuals studying or working in data science who are interested in seeing practical applications of machine learning concepts. This group values understanding the model's reasoning, performance metrics, and the balance of features in making predictions.

- **Entertainment Industry Professionals:** Producers, investors, marketers, and other film industry professionals seeking data-driven insights into commercial success factors. These users focus on practical applications and patterns revealed through the prediction process.
- **Casual Gamers:** General users looking for entertaining, educational games with competitive elements. This group prioritizes the game mechanics, scoring system, and overall engagement of the interface over technical details.
- **Educational Institutions:** Teachers and professors in film studies, business, or data science who can use the application as a teaching tool to demonstrate concepts of prediction, machine learning, or entertainment economics.

1.3.2 Detailed User Personas

To better understand the needs of our target users, we developed several detailed personas:

Maria T. - The Film Buff

- **Background:** 34-year-old film enthusiast who watches 100+ movies annually and participates in several online film communities
- **Technical Level:** Moderate; comfortable with technology but not technically focused
- **Goals:** Test her movie knowledge, discover patterns in successful films, compete with friends
- **Preferred Features:** Extensive movie database, detailed feedback on predictions, social sharing

Alex K. - The Data Science Student

- **Background:** 22-year-old computer science major studying machine learning and AI
- **Technical Level:** Advanced; familiar with programming and ML concepts
- **Goals:** Understand model architecture, examine feature importance, analyze prediction errors
- **Preferred Features:** Model explanations, confidence scores, feature breakdowns

Robert L. - The Film Producer

- **Background:** 45-year-old independent film producer with 15 years of industry experience
- **Technical Level:** Basic; focused on business applications rather than technical details
- **Goals:** Gain insights for investment decisions, identify success patterns by genre
- **Preferred Features:** Genre-specific statistics, budget vs. success analysis, market trends

Emma C. - The Casual User

- **Background:** 28-year-old occasional moviegoer who enjoys quiz games
- **Technical Level:** Basic; uses everyday technology but has no specialized knowledge
- **Goals:** Entertainment, learning interesting facts about movies, killing time pleasantly
- **Preferred Features:** Simple interface, quick game sessions, entertaining feedback

1.3.3 User Needs and Application Requirements

Based on these target audiences and personas, we identified several key requirements:

- **Multi-layered Information:** Present both simple results for casual users and detailed analytics for technical users
- **Varying Difficulty Levels:** Accommodate different expertise levels through game mode selection
- **Educational Components:** Include informative feedback that explains prediction factors
- **Engaging Interface:** Balance educational content with game elements that maintain interest
- **Domain-specific Insights:** Highlight patterns relevant to film industry professionals
- **Technical Transparency:** Provide sufficient information about the model for data science users

These audience considerations directly influenced the application's architecture, interface design, game mechanics, and feedback systems, as detailed in subsequent chapters.

Chapter 2

Project Architecture

2.1 System Architecture Overview

The Movie Success Predictor implements a modern, distributed architecture that separates concerns between client-side rendering, server-side processing, and external service integration. The system follows industry best practices including RESTful API design, stateless communication, and component-based frontend development.

2.1.1 High-Level Architecture

2.1.2 Architecture Components and Interactions

The Movie Success Predictor follows a client-server architecture with multiple interconnected components:

- **Frontend (Client):**
  - A React application with TypeScript providing the presentation layer
  - Component hierarchy for modular UI development and maintenance
  - Local state management using React hooks for user interaction and game progression
  - Dedicated service modules for API communication with typed interfaces
  - Styled with Tailwind CSS for responsive design across devices
  - Enhanced with animations via Framer Motion for engaging user experience
- **Backend (Server):**
  - Flask-based RESTful API handling client requests
  - CORS-enabled endpoints to facilitate secure cross-origin requests
  - Data processing pipeline with pandas for feature manipulation
  - Integration layer with the machine learning prediction service
  - Caching mechanisms for optimizing repeated requests

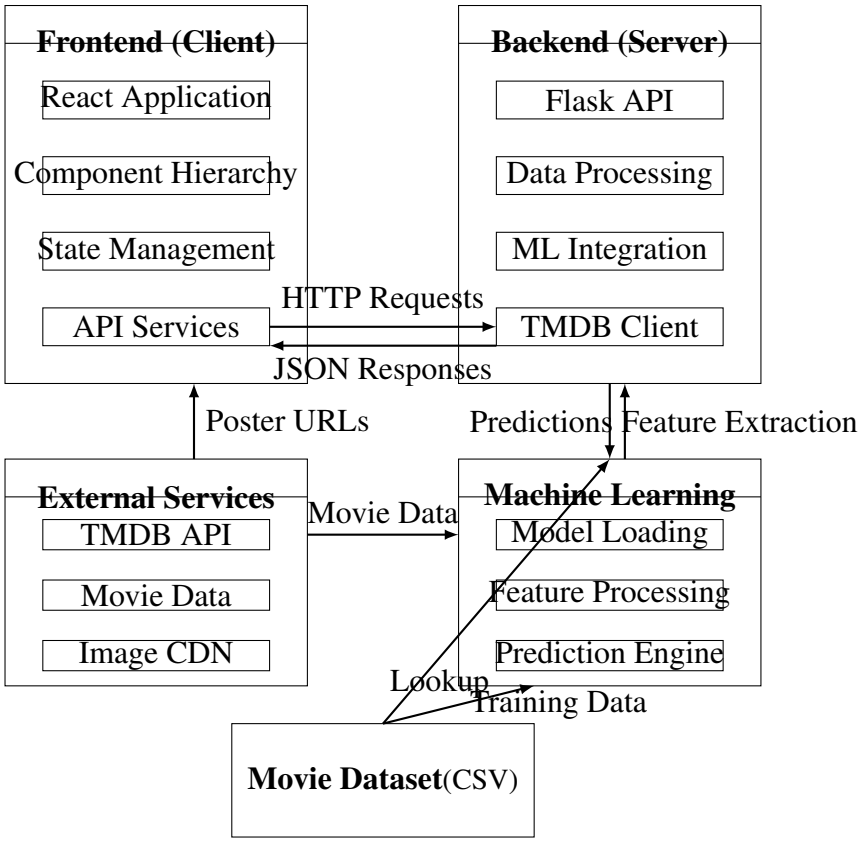

Figure 2.1: Detailed system architecture of the Movie Success Predictor

  - Error handling and logging systems for robust operation
  - Client interface for external TMDB API communication

- **Machine Learning Component:**
  - Pre-trained logistic regression model loaded at application startup
  - Feature processing pipeline ensuring consistent encoding
  - Prediction engine generating classification results with confidence scores
  - Model persistence using joblib for efficient loading
  - Consistent feature representation across training and inference

- **External Services:**
  - Integration with The Movie Database (TMDB) API for current movie data
  - Access to high-quality poster images through TMDB's CDN
  - API key management for secure external service communication
  - Rate limiting and error handling for external API resilience

- **Data Storage:**

  - CSV-based movie dataset containing historical movie information
  - Serialized model files stored on the server filesystem
  - Feature column information saved separately for consistent processing

2.1.3 Communication Protocols

The system components communicate using the following protocols and patterns:

- **Frontend to Backend:** RESTful HTTP requests using JSON for data interchange
- **Backend to ML Model:** In-memory function calls for efficient prediction
- **Backend to External APIs:** HTTP requests with API key authentication
- **Error Handling:** HTTP status codes with descriptive error messages
- **Data Validation:** Type checking on both client and server sides

2.1.4 Architectural Design Patterns

The application implements several key design patterns to ensure maintainability and scalability:

- **Model-View-Controller (MVC):** Separation of data model, business logic, and presentation
- **Component-Based Architecture:** Modular UI elements with encapsulated functionality
- **Service Layer:** Abstracted API communication through dedicated service modules
- **Repository Pattern:** Centralized data access for the movie database
- **Factory Pattern:** Dynamic creation of game modes and filter options
- **Observer Pattern:** Event-driven updates for UI state changes
- **Strategy Pattern:** Interchangeable algorithms for different game modes

2.1.5 Deployment Architecture

The application is designed for flexible deployment scenarios:

- **Development:** Local Vite development server with hot module replacement for frontend, Flask development server for backend
- **Production:** Static file hosting for compiled frontend assets, containerized backend service with gunicorn or uWSGI
- **Scaling:** Horizontally scalable backend with stateless request processing
- **Caching:** Client-side caching of static assets, server-side caching of frequent movie requests

2.2 Technology Stack and Implementation Details

The project integrates a diverse set of modern technologies across its frontend, backend, and machine learning components. Each technology was selected for specific capabilities that address the project's requirements for performance, maintainability, and user experience.

2.2.1 Frontend Technologies

Core Frameworks and Languages

- **React 18:** Modern JavaScript library for building user interfaces with component-based architecture, virtual DOM for efficient rendering, and hooks for state management. Used throughout the application for creating reusable UI components and maintaining consistent user experience.
- **TypeScript 4.9:** Statically typed superset of JavaScript that enhances code quality and developer experience through type checking and IDE integration. All frontend code is written in TypeScript with comprehensive interface definitions for type safety.
- **Vite 4.3:** Next-generation frontend build tool that offers significantly faster development server startup and hot module replacement compared to traditional bundlers. Configured with custom settings for optimized production builds.
- **HTML5/CSS3:** Modern markup and styling standards used for semantic structure and advanced visual effects. Support for the latest features like CSS variables, flexbox, and grid layouts.

UI and Styling

- **Tailwind CSS 3.3:** Utility-first CSS framework enabling rapid UI development with minimal custom CSS. Configured with project-specific theme extending base components with custom colors, fonts, and design elements.
- **PostCSS:** Tool for transforming CSS with JavaScript plugins, used for processing Tailwind directives and optimizing CSS output. Configured with autoprefixer for cross-browser compatibility.
- **Framer Motion 10.12:** Production-ready animation library for React that simplifies the implementation of complex UI animations and transitions. Used for card animations, feedback effects, and page transitions.
- **Heroicons:** SVG icon collection designed specifically for Tailwind CSS projects, providing consistent, accessible, and customizable icons throughout the interface.

State Management and Data Handling

- **React Hooks:** Built-in React features like useState, useEffect, useContext, and custom hooks used for local and global state management without additional libraries.
- **Axios 1.4:** Promise-based HTTP client for making API requests to the backend with features like request/response interception, automatic JSON transformation, and client-side error handling. Configured with custom instance for the backend API with default headers and base URL.
- **React-Use:** Collection of essential React hooks providing solutions for common UI needs like window size detection, intersection observation, and media queries.
- **TypeScript Interfaces:** Custom type definitions for all data structures ensuring type safety across component boundaries and API calls.

User Experience Enhancements

- **React-Confetti:** Library for creating configurable confetti animations to celebrate correct answers and achievements.
- **HTML5 Audio API:** Native browser API used for implementing sound effects for user interactions, feedback, and game events.
- **CSS Transitions and Animations:** Custom animations for loading states, error handling, and interactive elements enhancing user feedback and engagement.
- **Responsive Design:** Mobile-first approach ensuring proper display and functionality across devices of different screen sizes.

Development and Tooling

- **ESLint:** JavaScript linter configured with React and TypeScript-specific rules to enforce code quality and consistency.
- **TypeScript Compiler:** Configured with strict type-checking options for maximum type safety across the codebase.
- **npm:** Package manager for dependency management and script execution.

2.2.2 Backend Technologies

Web Framework and API

- **Flask 2.3:** Lightweight Python web framework used for building the RESTful API endpoints. Selected for its simplicity, flexibility, and compatibility with machine learning libraries.

- **Flask-CORS 3.0:** Flask extension that handles Cross-Origin Resource Sharing, enabling secure communication between frontend and backend running on different origins.
- **Werkzeug:** WSGI web application library used by Flask for request/response handling, routing, and debugging.
- **dotenv:** Library for loading environment variables from .env files, used for configuration management including API keys and deployment settings.

Data Processing and Analysis

- **Pandas 2.0:** Powerful data manipulation and analysis library used for loading, processing, and filtering the movie dataset. Handles data transformation operations like one-hot encoding and feature normalization.
- **NumPy 1.24:** Fundamental package for scientific computing with Python, providing support for large, multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays. Used for numerical operations in data preprocessing.
- **Python Standard Library:** Built-in modules like random (for random movie selection), os (for file path handling), and json (for data serialization).

Machine Learning

- **Scikit-learn 1.2:** Machine learning library providing simple and efficient tools for predictive data analysis. Used for implementing the Logistic Regression model and preprocessing pipelines.
- **Joblib 1.2:** Library for serializing and deserializing Python objects, particularly optimized for scientific computing. Used for saving and loading the trained model and feature columns.
- **GridSearchCV:** Scikit-learn utility for hyperparameter tuning through exhaustive grid search with cross-validation.

External API Integration

- **TMDbSimple:** Python wrapper for The Movie Database API providing convenient access to movie data and poster images.
- **Requests:** Simple HTTP library for making API calls to external services when needed beyond TMDbSimple functionality.

Development and Deployment

- **Python 3.9+:** Modern Python version with type hinting support and improved performance.
- **Virtual Environment:** Python's venv module for dependency isolation and environment management.
- **Requirements.txt:** Standard dependency specification file for Python projects.

2.2.3 Database and Storage

- **CSV Storage:** Flat file storage for the movie dataset (moviesDb.csv) containing cleaned and preprocessed movie data.
- **Joblib Storage:** Binary file storage for the trained machine learning model and associated metadata.
- **File System:** Direct file system access for reading the dataset and model files.

2.2.4 API and Integration

- **RESTful API Design:** Consistent endpoint design following REST principles for intuitive and predictable API interactions.
- **JSON:** Standard data interchange format for all API communications between frontend, backend, and external services.
- **HTTP/HTTPS:** Standard web protocols for all network communications with proper error status codes.
- **TMDB API v3:** External API providing movie details, metadata, and poster images with authorization via API key.

2.2.5 Technology Selection Rationale

The technology stack was carefully selected based on several key considerations:

- **Developer Experience:** Tools like TypeScript, Vite, and Flask enhance development efficiency through features like type checking, fast reloading, and intuitive APIs.
- **Performance:** React's virtual DOM, Vite's optimized builds, and Flask's lightweight architecture ensure responsive performance across the application.
- **Maintainability:** Component-based design, strict typing, and modular architecture promote code quality and ease of maintenance.
- **Scalability:** The chosen technologies support future expansion through well-defined interfaces and separation of concerns.
- **User Experience:** Animation libraries, sound effects, and responsive design create an engaging and accessible user interface.
- **Integration Capabilities:** RESTful API design and standardized data formats enable seamless communication between system components.

2.3 Data Flow

1. User interacts with the frontend interface (selecting categories, making guesses)
2. Frontend sends requests to the Flask backend API
3. Backend processes requests and retrieves movie information from its database
4. For movie details and posters, the backend communicates with the TMDB API
5. For predictions, the backend uses the pre-trained logistic regression model
6. Results are sent back to the frontend for display to the user
7. Frontend updates the UI based on the response (showing feedback, updating score)

Chapter 3

Frontend Implementation

3.1 Frontend Architecture and Design Philosophy

The frontend of the Movie Success Predictor is built as a modern React single-page application (SPA) with TypeScript for enhanced type safety and developer experience. The architecture follows an opinionated component-based design focusing on modularity, reusability, and separation of concerns.

The application adheres to several key design principles:

- **Component-Based Design:** Breaking the UI into self-contained, reusable components that manage their own state and rendering
- **Separation of Concerns:** Clear distinction between UI components, business logic, and data services
- **Type Safety:** Comprehensive TypeScript typing throughout the codebase to prevent runtime errors
- **Responsive Design:** Mobile-first approach ensuring proper function across diverse devices
- **Progressive Enhancement:** Core functionality works without JavaScript, with enhanced features when available
- **Accessibility:** Following WCAG guidelines for inclusive user experience

3.1.1 Project Structure and Organization

The frontend codebase is organized following a feature-based and layered architecture pattern. This organization allows for better code navigation, maintainability, and scalability as the application grows.

```
frontend/
 src/                          # Source code directory
    components/               # Reusable UI components
       FeedbackMessage.tsx # User feedback display component
       FilterControls.tsx  # Category filter selection component
```

```
      Footer.tsx           # Global footer component
      MovieCard.tsx        # Movie display card component
      Navbar.tsx           # Navigation header component
      QuizControls.tsx     # Hit/Flop buttons component
      ScoreDisplay.tsx     # User score tracking component
   pages/                   # Top-level page components
      AboutPage.tsx        # About project information page
      Documentation.tsx    # Technical documentation page
      FeaturesPage.tsx     # Features showcase page
      LandingPage.tsx      # Welcome/entry page
      QuizPage.tsx         # Main quiz interaction page
      TeamPage.tsx         # Team members information page
   services/                # API service abstractions
      backendService.ts    # Flask backend API client
      tmdbService.ts       # TMDB API interactions
   hooks/                   # Custom React hooks
      useLocalStorage.ts   # Hook for persistent storage
   types/                   # TypeScript type definitions
      index.ts             # Shared type interfaces
   assets/                  # Static assets
      react.svg            # React logo asset
   utils/                   # Utility functions
      formatters.ts        # Text and data formatting helpers
      validators.ts        # Input validation helpers
   constants/               # Application constants
      gameConfig.ts        # Game configuration constants
   App.tsx                  # Root component with routing
   main.tsx                 # Application entry point
   index.css                # Global styles and Tailwind imports
   vite-env.d.ts            # Vite type declarations
public/                      # Public static assets
   sounds/                  # Audio effect files
      correct.mp3          # Sound for correct guesses
      gameover.mp3         # Sound for game completion
      start.mp3            # Sound for game initialization
      streak.mp3           # Sound for consecutive correct answers
      wrong.mp3            # Sound for incorrect guesses
   vite.svg                 # Application logo
index.html                   # HTML entry point
package.json                 # NPM dependencies and scripts
package-lock.json            # Locked dependency versions
tsconfig.json                # TypeScript configuration
tsconfig.app.json            # App-specific TypeScript config
tsconfig.node.json           # Node-specific TypeScript config
postcss.config.js            # PostCSS configuration for Tailwind
tailwind.config.js           # Tailwind CSS customization
vite.config.ts               # Vite bundler configuration
eslint.config.js             # ESLint code quality rules
```

Listing 3.1: Detailed Frontend Directory Structure

3.1.2 Design System and Visual Language

The application implements a consistent design system based on a carefully selected color palette, typography, and component styling guidelines. The design system ensures visual coherence across the application while enabling component reusability.

Color Scheme

- **Primary Colors:** Deep blues (`#1E40AF`, `#3B82F6`) for primary actions and emphasis
- **Secondary Colors:** Rich purples (`#7C3AED`, `#8B5CF6`) for secondary elements
- **Accent Colors:** Vibrant teals (`#0D9488`, `#14B8A6`) for highlighting and special elements
- **Semantic Colors:** Green (`#10B981`) for success, Red (`#EF4444`) for errors, Yellow (`#F59E0B`) for warnings
- **Neutral Colors:** Grayscale range from white (`#FFFFFF`) to very dark (`#111827`) for text, backgrounds, and borders
- **Game Mode Colors:** Custom gradients for each game difficulty level

Typography System

- **Primary Font:** Inter (sans-serif) for general text
- **Display Font:** Montserrat for headings and emphasis
- **Type Scale:** Modular scale based on 1.25 ratio
- **Font Weights:** 400 (normal), 500 (medium), 600 (semibold), 700 (bold)
- **Line Heights:** 1.5 for body text, 1.2 for headings

Component Design Principles

- **Consistency:** Similar components maintain consistent styling and behavior
- **Feedback:** All interactive elements provide visual feedback on hover/focus/active states
- **Motion:** Purposeful animations enhance understanding of state changes
- **Accessibility:** Sufficient contrast ratios, focus indicators, and ARIA attributes
- **Responsiveness:** Fluid layouts adapting to different viewport sizes

Component	Variants	States
Button	Primary, Secondary, Tertiary	Default, Hover, Active, Disabled
Card	Standard, Interactive, Highlighted	Default, Hover, Selected
Input	Text, Select, Radio	Default, Focus, Error, Disabled

Figure 3.1: Sample of design system component specifications

3.2 Core UI Components

The frontend is composed of multiple reusable UI components, each responsible for a specific aspect of the user interface. These components follow a compositional pattern where complex UI elements are built from simpler, focused components with clear interfaces. Below we detail the most critical components in the application.

3.2.1 MovieCard Component

The MovieCard component is responsible for displaying movie information and poster images, forming the central visual element of the quiz experience.

Implementation

```
import React, { useState } from 'react';
import { motion } from 'framer-motion';

interface MovieCardProps {
  title: string;
  posterUrl: string;
}

const MovieCard = ({ title, posterUrl }: MovieCardProps) => {
  const [imageLoaded, setImageLoaded] = useState(false);
  const [imageError, setImageError] = useState(false);

  const handleImageLoad = () => {
    setImageLoaded(true);
  };

  const handleImageError = () => {
    setImageError(true);
  };

  return (
    <motion.div
      initial={{ opacity: 0, scale: 0.95 }}
      animate={{ opacity: 1, scale: 1 }}
      transition={{ duration: 0.3 }}
      className="relative w-full max-w-sm"
    >
```

```
      <div className="aspect-[2/3] overflow-hidden rounded-xl bg-dark-800 relative">
        {/* Loading skeleton */}
        {!imageLoaded && !imageError && (
          <div className="absolute inset-0 bg-dark-700 animate-pulse" />
        )}

        {/* Movie poster */}
        {!imageError ? (
          <motion.img
            src={posterUrl}
            alt={title}
            onLoad={handleImageLoad}
            onError={handleImageError}
            initial={{ opacity: 0 }}
            animate={{ opacity: imageLoaded ? 1 : 0 }}
            transition={{ duration: 0.3 }}
            className="w-full h-full object-cover"
          />
        ) : (
          /* Fallback for loading errors */
          <div className="w-full h-full flex items-center justify-center bg-dark-700">
            <svg className="w-16 h-16" fill="none" stroke="currentColor" viewBox="0 0 24 24">
              <path d="M7 4v16M17 4v16M3 8h4m10 0h4M3 12h18M3 16h4m10 0h4M4 20h16a1 1 0 001-1V5a1 1 0 00-1-1H4a1 1 0 00-1 1v14a1 1 0 001 1z" />
            </svg>
          </div>
        )}

        {/* Title overlay with gradient */}
        <div className="absolute inset-0 bg-gradient-to-t from-dark-900 via-transparent to-transparent" />
        <motion.div
          initial={{ opacity: 0, y: 20 }}
          animate={{ opacity: 1, y: 0 }}
          transition={{ delay: 0.2 }}
          className="absolute bottom-0 left-0 right-0 p-6"
        >
          <h2 className="text-2xl font-display font-bold text-white text-center">{title}</h2>
        </motion.div>
      </div>
    </motion.div>
  );
};

export default MovieCard;
```

Listing 3.2: MovieCard Component Implementation Excerpt

Features

- **Progressive Loading:** Displays a loading skeleton while the poster image is loading
- **Error Handling:** Gracefully displays a fallback when images fail to load
- **Animations:** Smooth entrance animations using Framer Motion
- **Accessibility:** Proper alt text and semantic HTML structure
- **Responsive Design:** Maintains aspect ratio across screen sizes
- **Visual Enhancement:** Gradient overlay ensures title readability over any poster

3.2.2 QuizControls Component

The QuizControls component provides the primary interactive elements of the quiz, allowing users to submit their predictions about a movie's success.

Implementation

```
import React from 'react';
import { motion } from 'framer-motion';

interface QuizControlsProps {
  onGuess: (guess: 'Hit' | 'Flop') => void;
  disabled: boolean;
}

const QuizControls: React.FC<QuizControlsProps> = ({ onGuess, disabled }) => {
  return (
    <div className="flex justify-center space-x-8 py-6">
      <motion.button
        whileHover={{ scale: disabled ? 1 : 1.05 }}
        whileTap={{ scale: disabled ? 1 : 0.95 }}
        className={`
          px-8 py-4 rounded-xl font-bold text-xl
          bg-gradient-to-br from-green-500 to-emerald-600
          text-white shadow-lg shadow-green-500/30
          transition-all duration-200
          ${disabled ? 'opacity-50 cursor-not-allowed' : 'hover:shadow-xl hover:shadow-green-500/40'}
        `}
        onClick={() => !disabled && onGuess('Hit')}
        disabled={disabled}
        aria-label="Guess Hit"
      >
        HIT
      </motion.button>

      <motion.button
```

```
        whileHover={{ scale: disabled ? 1 : 1.05 }}
        whileTap={{ scale: disabled ? 1 : 0.95 }}
        className={`
          px-8 py-4 rounded-xl font-bold text-xl
          bg-gradient-to-br from-red-500 to-rose-600
          text-white shadow-lg shadow-red-500/30
          transition-all duration-200
          ${disabled ? 'opacity-50 cursor-not-allowed' : 'hover:shadow-xl
    hover:shadow-red-500/40'}
        `}
        onClick={() => !disabled && onGuess('Flop')}
        disabled={disabled}
        aria-label="Guess Flop"
      >
        FLOP
      </motion.button>
    </div>
  );
};

export default QuizControls;
```

Listing 3.3: QuizControls Component Implementation Excerpt

Features

- **Clear Visual Distinction:** Color-coded buttons for different predictions
- **Interactive Feedback:** Scale and shadow effects on hover and tap
- **Disabled State:** Visual indication when controls are not available
- **Accessibility:** Proper ARIA labels and disabled attributes

3.2.3 ScoreDisplay Component

This component visualizes the user's performance metrics during the quiz session, providing real-time feedback on their progress.

Implementation

```
import React from 'react';
import { motion } from 'framer-motion';

interface ScoreDisplayProps {
  score: number;
  totalQuestionsAnswered: number;
  questionsRemaining?: number;
}

const ScoreDisplay: React.FC<ScoreDisplayProps> = ({
```

```
  score,
  totalQuestionsAnswered,
  questionsRemaining
}) => {
  // Calculate percentage for progress bar
  const progressPercentage = questionsRemaining !== undefined &&
   questionsRemaining > 0
    ? Math.round(((totalQuestionsAnswered) / (totalQuestionsAnswered +
   questionsRemaining)) * 100)
    : 0;

  return (
    <motion.div
      initial={{ opacity: 0, y: -20 }}
      animate={{ opacity: 1, y: 0 }}
      transition={{ duration: 0.5 }}
      className="bg-dark-800/80 backdrop-blur-sm rounded-xl p-4 shadow-lg"
    >
      <div className="flex items-center justify-between">
        <div className="text-center px-4">
          <p className="text-sm text-gray-400 uppercase tracking-wide">
   Score</p>
          <p className="text-2xl font-bold text-white">{score}</p>
        </div>

        <div className="text-center px-4">
          <p className="text-sm text-gray-400 uppercase tracking-wide">
   Answered</p>
          <p className="text-2xl font-bold text-white">{
   totalQuestionsAnswered}</p>
        </div>

        {questionsRemaining !== undefined && (
          <div className="text-center px-4">
            <p className="text-sm text-gray-400 uppercase tracking-wide">
   Remaining</p>
            <p className="text-2xl font-bold text-white">{
   questionsRemaining}</p>
          </div>
        )}
      </div>

      {questionsRemaining !== undefined && (
        <div className="mt-2 bg-dark-700 rounded-full h-2 overflow-hidden"
   >
          <motion.div
            className="h-full bg-gradient-to-r from-blue-500 to-purple-500
   "
            initial={{ width: '0%' }}
            animate={{ width: `${progressPercentage}%` }}
            transition={{ duration: 0.5 }}
          />
        </div>
      )}
```

```
    </motion.div>
  );
};

export default ScoreDisplay;
```

Listing 3.4: ScoreDisplay Component Implementation Excerpt

Features

- **Multiple Metrics:** Displays score, questions answered, and questions remaining
- **Visual Feedback:** Animated progress bar for game completion tracking
- **Entrance Animation:** Smooth appearance with y-axis movement
- **Conditional Rendering:** Adapts to game modes with or without question limits

3.2.4 FeedbackMessage Component

This component delivers visual and textual feedback to users after they make a guess, enhancing the learning experience by explaining the outcome.

Implementation

```
import React from 'react';
import { motion } from 'framer-motion';
import { CheckCircleIcon, XCircleIcon } from '@heroicons/react/24/outline';

interface FeedbackMessageProps {
  message: string;
  isCorrect: boolean | null;
}

const FeedbackMessage: React.FC<FeedbackMessageProps> = ({ message, isCorrect }) => {
  if (!message) return null;

  const bgColorClass = isCorrect
    ? 'bg-gradient-to-r from-green-500/20 to-emerald-500/20 border-green-500/50'
    : 'bg-gradient-to-r from-red-500/20 to-rose-500/20 border-red-500/50';

  const textColorClass = isCorrect ? 'text-green-500' : 'text-rose-500';
  const Icon = isCorrect ? CheckCircleIcon : XCircleIcon;

  return (
    <motion.div
      initial={{ opacity: 0, y: 20, scale: 0.95 }}
      animate={{ opacity: 1, y: 0, scale: 1 }}
```

```
      exit={{ opacity: 0, scale: 0.95 }}
      transition={{ duration: 0.4, type: 'spring' }}
      className={`p-4 rounded-xl border ${bgColorClass} shadow-lg backdrop-blur-sm max-w-xl mx-auto`}
    >
      <div className="flex items-start">
        <Icon className={`${textColorClass} w-6 h-6 mr-3 flex-shrink-0 mt-0.5`} />
        <div>
          <p className={`${textColorClass} font-bold text-lg`}>
            {isCorrect ? 'Correct!' : 'Not quite!'}
          </p>
          <p className="text-white/90 mt-1">{message}</p>
        </div>
      </div>
    </motion.div>
  );
};

export default FeedbackMessage;
```

Listing 3.5: FeedbackMessage Component Implementation Excerpt

Features

- **Contextual Display:** Shows different styles based on correctness of guess
- **Branded Icons:** Uses Heroicons for visual indicators of result
- **Rich Animation:** Spring physics for natural-feeling entrance and exit
- **Conditional Rendering:** Only appears when feedback is available

3.2.5 FilterControls Component

This component provides controls for filtering the quiz content by various movie attributes, allowing users to customize their experience.

Implementation

```
import React from 'react';
import { motion } from 'framer-motion';
import type { FilterOptions, GetMovieParams } from '../services/backendService';

interface FilterControlsProps {
  options: FilterOptions | null;
  selectedFilters: GetMovieParams;
  onFilterChange: (filterType: keyof GetMovieParams, value: string) => void;
  isLoading: boolean;
```

```
}

const FilterControls: React.FC<FilterControlsProps> = ({
  options,
  selectedFilters,
  onFilterChange,
  isLoading
}) => {
  if (!options) return null;

  return (
    <motion.div
      initial={{ opacity: 0 }}
      animate={{ opacity: 1 }}
      className="grid grid-cols-1 sm:grid-cols-3 gap-4 w-full max-w-4xl mx-auto"
    >
      {/* Genre Filter */}
      <div className="w-full">
        <label className="block text-sm font-medium text-gray-300 mb-1">
          Genre
        </label>
        <select
          value={selectedFilters.genre || ''}
          onChange={(e) => onFilterChange('genre', e.target.value)}
          disabled={isLoading}
          className="block w-full rounded-lg bg-dark-700 border border-dark-600 text-white py-2 px-3 focus:ring-2 focus:ring-primary-500 focus:border-primary-500"
        >
          <option value="">All Genres</option>
          {options.genres.map((genre) => (
            <option key={genre} value={genre}>
              {genre}
            </option>
          ))}
        </select>
      </div>

      {/* Country Filter */}
      <div className="w-full">
        <label className="block text-sm font-medium text-gray-300 mb-1">
          Country
        </label>
        <select
          value={selectedFilters.country || ''}
          onChange={(e) => onFilterChange('country', e.target.value)}
          disabled={isLoading}
          className="block w-full rounded-lg bg-dark-700 border border-dark-600 text-white py-2 px-3 focus:ring-2 focus:ring-primary-500 focus:border-primary-500"
        >
          <option value="">All Countries</option>
          {options.countries.map((country) => (
```

```
            <option key={country} value={country}>
              {country}
            </option>
          ))}
        </select>
      </div>

      {/* Certification Filter */}
      <div className="w-full">
        <label className="block text-sm font-medium text-gray-300 mb-1">
          Certification
        </label>
        <select
          value={selectedFilters.certification || ''}
          onChange={(e) => onFilterChange('certification', e.target.value)
    }
          disabled={isLoading}
          className="block w-full rounded-lg bg-dark-700 border border-
    dark-600 text-white py-2 px-3 focus:ring-2 focus:ring-primary-500 focus
    :border-primary-500"
        >
          <option value="">All Ratings</option>
          {options.certifications.map((cert) => (
            <option key={cert} value={cert}>
              {cert}
            </option>
          ))}
        </select>
      </div>
    </motion.div>
  );
};

export default FilterControls;
```

Listing 3.6: FilterControls Component Implementation Excerpt

Features

- **Dynamic Options:** Populates filter dropdowns based on available data
- **Multi-dimension Filtering:** Allows filtering by genre, country, and certification
- **Responsive Layout:** Adapts from single column on mobile to three columns on larger screens
- **Loading State:** Disables controls during data loading

3.2.6 Core Navigation Components

The application includes several navigation and structural components that provide consistent user experience across different pages.

Navbar Component

- Provides global navigation with links to all major sections
- Implements responsive design with mobile menu toggle
- Indicates current section with active state styling
- Includes project branding and logo

Footer Component

- Displays credits, copyright information, and team details
- Includes links to related resources and documentation
- Provides social media links and contact information
- Maintains consistent branding and styling with the rest of the application

3.3 Pages and Routing

1. **QuizPage:** Main interactive quiz interface
2. **LandingPage:** Introduction and game start
3. **AboutPage:** Information about the project
4. **TeamPage:** Details about team members
5. **FeaturesPage:** Features of the application
6. **Documentation:** Technical documentation

3.4 State Management

The application uses React's built-in state management features:

- **useState:** For component-level state (current movie, score, feedback)
- **useEffect:** For side effects like API calls and timer management
- **useCallback:** For performance optimization with memoized functions
- **useRef:** For persistent values that don't cause re-renders (timers, audio)

3.5 API Services

Two main service files handle external API communication:

- **backendService.ts:** Communicates with the Flask backend
- **tmdbService.ts:** Manages TMDB API interactions for movie posters

3.6 User Interface

The user interface is designed to be modern, engaging, and responsive. It features:

- Game mode selection with difficulty levels
- Category filters for movie selection
- Interactive movie cards with poster displays
- Animated feedback messages
- Visual effects for correct/incorrect answers
- Sound effects for enhanced engagement
- Score tracking and game summary

Chapter 4

Backend Implementation

4.1 Backend Architecture

The backend is built with Flask, a lightweight Python web framework. It serves as an API that processes requests from the frontend, interacts with the machine learning model, and communicates with external services.

4.1.1 Directory Structure

```
backend/
 app.py                          # Main Flask application
 train_model.py                  # Script for training the ML model
 data/
    moviesDb.csv                # Movie dataset
 models/
    logistic_regression_model.joblib  # Trained model
    model_columns.joblib        # Feature columns for the model
 requirements.txt                # Python dependencies
```

Listing 4.1: Backend Directory Structure

4.2 API Endpoints

- **GET /api/quiz/filter-options:** Returns available filters (genres, countries, certifications)
- **GET /api/quiz/next-movie:** Returns a random movie for the quiz, with optional filters
- **POST /api/quiz/submit-guess:** Processes user's guess and returns feedback

4.3 Data Processing

1. Data is loaded from the CSV file into a pandas DataFrame
2. Missing values are handled appropriately (filled with "Unknown" for categorical variables)

3. Filter parameters from the frontend are applied to query specific movies
4. Random sampling is used to select movies for the quiz
5. Movie information is formatted and returned to the frontend

4.4 Error Handling

The backend implements robust error handling:

- Missing data file errors
- Model loading failures
- Invalid request parameters
- No movies matching filter criteria
- TMDB API connection issues

4.5 Integration with TMDB API

The backend uses TMDbSimple, a Python wrapper for The Movie Database API, to:

- Fetch movie posters
- Validate movie information
- Provide high-quality visual content for the frontend

Chapter 5

Machine Learning Component

5.1 Dataset Overview

5.1.1 Data Source

The project uses a movie dataset derived from The Movie Database (TMDB) with the following characteristics:

- Original source: Kaggle's TMDB 5000 Movies dataset
- Extensively cleaned and preprocessed for the project
- Contains 5,129 movie entries with various attributes

5.1.2 Features

- **Numerical features:** budget, runtime, year, vote_average, vote_count
- **Categorical features:** genre, country, certification
- **Target variable:** success (boolean - True for hit, False for flop)

5.2 Data Preprocessing

5.2.1 Cleaning Steps

1. Removing duplicates
2. Filtering out movies with missing critical information
3. Converting dates to extract year
4. Creating the 'success' column based on revenue/budget ratio
5. Extracting primary genre and country from lists

6. Removing low-frequency categories
7. Standardizing categorical values

5.2.2 Feature Engineering

- **Success definition:** A movie is considered successful (hit) if revenue >= budget * 2
- **Year extraction:** Extracted from release_date
- **Categorical encoding:** One-hot encoding for genre, country, and certification

5.3 Model Selection

5.3.1 Model Evaluation

Several classification algorithms were evaluated:

- Logistic Regression
- K-Nearest Neighbors (KNN)
- Decision Tree
- Random Forest

5.3.2 Model Performance

Through GridSearchCV with 10-fold cross-validation, logistic regression emerged as the best model:

- **Best hyperparameters:** C=1000, solver='newton-cg'
- **Performance metrics:** Accuracy, precision, recall, F1-score

5.4 Model Training and Saving

1. Data preparation with one-hot encoding
2. Model initialization with optimized hyperparameters
3. Training on the entire dataset
4. Serialization of the model using joblib
5. Saving model columns for consistent feature representation

5.5 Model Deployment

- Model is loaded during Flask application startup
- Prediction requests process movie features to match training format
- One-hot encoding is applied consistently with training data
- Prediction probabilities provide confidence metrics
- Feedback includes model prediction, actual result, and user correctness

Chapter 6

Data Flow and Integration

6.1 Frontend to Backend Communication

6.1.1 API Request Flow

1. User initiates action (starting game, selecting filters, making guess)
2. Frontend forms API request with appropriate parameters
3. Axios sends HTTP request to backend endpoint
4. Backend processes request and forms response
5. Frontend receives and processes response
6. UI updates to reflect new state

6.1.2 Key Data Interfaces

```
interface Movie {
  id: string;
  title: string;
  posterUrl?: string;
}

interface FilterOptions {
  genres: string[];
  countries: string[];
  certifications: string[];
}

interface GetMovieParams {
  genre?: string;
  country?: string;
```

```
  certification?: string;
}
```

6.2 Backend Data Processing

6.2.1 Movie Selection Process

```python
import pandas as pd
import random

def get_next_movie(filters=None):
    df = pd.read_csv('data/moviesDb.csv')
    if filters:
        for key, value in filters.items():
            if value:
                df = df[df[key] == value]
    if not df.empty:
        movie = df.sample(1).iloc[0]
        return {
            'id': movie['id'],
            'title': movie['title'],
            'poster': movie.get('poster_url')
        }
    return None
```

Listing 6.1: Backend Movie Selection Logic

6.3 Prediction and Feedback Flow

- User submits guess ('Hit' or 'Flop') for current movie
- Frontend sends movie ID and guess to backend
- Backend retrieves movie data from database
- Features are processed to match model's expected format
- Model makes prediction and determines actual result
- Response contains user's guess, model prediction, actual result
- Frontend displays feedback with animations and sound effects
- Score is updated and next question is prepared

Chapter 7

Detailed File and Module Explanation

7.1 Frontend Files

7.1.1 Main Components

- **App.tsx:** Root component that sets up routing
- **QuizPage.tsx:** Core game logic and state management
- **MovieCard.tsx:** Visual presentation of movie information
- **QuizControls.tsx:** User interaction controls for guessing

7.1.2 Service Files

- **backendService.ts:** Handles all backend API calls
- **tmdbService.ts:** Manages TMDB API interactions and poster URLs

7.2 Backend Files

7.2.1 Core Files

- **app.py:** Main Flask application with API endpoints

- **train**$_{m}odel.py$: $Script for training the logistic regression model$

7.2.2 Data and Model Files

- **moviesDb.csv:** Cleaned movie dataset

- **logistic**$_{r}egression_{m}odel.joblib$: $Serialized trained model$

7.3 Key Dependencies

- **React:** UI library for component-based development
- **Flask:** Lightweight web framework for the backend
- **Pandas:** Data manipulation library
- **Scikit-learn:** Machine learning toolkit

Chapter 8

Step-by-Step Application Workflow

8.1 User Interaction Flow

1. User visits the application landing page
2. User selects game mode (Chill Mode, Quick Five, Speed Demon, etc.)
3. Optionally, user selects movie categories (genre, country, certification)
4. Game begins and displays first movie with poster and title
5. User guesses whether the movie is a "Hit" or "Flop"
6. System provides feedback on the guess
7. Process repeats until game end condition (questions limit or time limit)
8. Final score and performance summary are displayed

8.2 Backend Processing Flow

1. Receive request for a new movie with optional filters
2. Apply filters to the movie dataset
3. Randomly select a movie from filtered results
4. Fetch poster from TMDB API if available
5. Return movie information to frontend
6. Receive guess submission with movie ID
7. Process movie features for model input
8. Get model prediction and actual result

9. Generate feedback message with results
10. Return complete response to frontend

8.3 Data Processing and Prediction Steps

1. Extract movie features from database
2. Handle missing values in features
3. Apply one-hot encoding to categorical features
4. Ensure feature columns match training data
5. Pass processed features to model for prediction
6. Get prediction probability for "Hit" class
7. Compare model prediction with actual outcome
8. Compare user guess with actual outcome
9. Generate detailed feedback message

Chapter 9

Conclusion and Summary

9.1 Project Achievements

The Movie Success Predictor project successfully:

- Built an end-to-end machine learning application with web interface
- Implemented a logistic regression model for movie success prediction
- Created an engaging interactive quiz format for users
- Integrated external API services for enhanced content
- Applied responsive design principles for cross-device compatibility
- Implemented sound effects and animations for user engagement

9.2 Challenges Faced

- Handling missing data in the movie dataset
- Ensuring consistent feature representation between training and prediction
- Managing cross-origin requests between frontend and backend
- Optimizing model performance for real-time predictions
- Creating a responsive and visually appealing user interface
- Implementing game logic with timing and scoring

9.3 Future Enhancements

- User authentication and persistent high scores
- More advanced machine learning models (ensemble methods)
- Additional movie features and metadata
- Expanded game modes and challenges
- Social sharing capabilities
- Mobile application version

Chapter 10

Appendices

10.1 Installation and Setup

10.1.1 Prerequisites

- Node.js and npm for the frontend
- Python 3.7+ for the backend
- TMDB API key for movie data

10.1.2 Frontend Setup

```
# Navigate to frontend directory
cd frontend

# Install dependencies
npm install

# Start development server
npm run dev
```

Listing 10.1: Frontend Setup Instructions

10.1.3 Backend Setup

```
# Navigate to backend directory
cd backend

# Create virtual environment
python -m venv venv

# Activate virtual environment
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install dependencies
```

```
pip install -r requirements.txt

# Train model (if needed)
python train_model.py

# Start Flask server
python app.py
```

Listing 10.2: Backend Setup Instructions

10.2 API Documentation

10.2.1 Filter Options Endpoint

- **URL:** /api/quiz/filter-options
- **Method:** GET
- **Response:** JSON object with genres, countries, and certifications lists

10.2.2 Next Movie Endpoint

- **URL:** /api/quiz/next-movie
- **Method:** GET
- **Parameters:** genre, country, certification (all optional)
- **Response:** JSON object with movie ID, title, and poster URL

10.2.3 Submit Guess Endpoint

- **URL:** /api/quiz/submit-guess
- **Method:** POST
- **Body:** JSON with movieId and guess ('Hit' or 'Flop')
- **Response:** JSON with user guess, model prediction, actual result, and feedback message

10.3 Dataset Details

- **Records:** 5,129 movies
- **Time period:** Movies from 1970 to recent years

- **Key fields:** id, title, budget, revenue, runtime, year, success, genre, certification$_U S, vote_a verage, vote_c ount, countr$y

10.4 Code Snippets and Examples

10.4.1 Model Training

```python
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
import joblib
import pandas as pd

# Load and preprocess data
df = pd.read_csv('data/moviesDb.csv')
# ... (data preprocessing steps)

# Define features and target
X = df.drop('success', axis=1)
y = df['success']

# Initialize and train model
param_grid = {'C': [0.1, 1, 10, 100, 1000], 'solver': ['newton-cg']}
model = GridSearchCV(LogisticRegression(), param_grid, cv=10)
model.fit(X, y)

# Save model
joblib.dump(model.best_estimator_, 'models/logistic_regression_model.
    joblib')
```

Listing 10.3: Logistic Regression Model Training

10.5 References

- The Movie Database (TMDB) - `https://www.themoviedb.org/`
- React Documentation - `https://reactjs.org/docs/getting-started.html`
- Flask Documentation - `https://flask.palletsprojects.com/`
- Scikit-learn Documentation - `https://scikit-learn.org/stable/`
- Tailwind CSS Documentation - `https://tailwindcss.com/docs`