Judging a Book by Its Cover: A Deep Learning Approach

Antonino Pio Lupo

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work, and including any code produced using generative AI systems. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion, or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

Can a book be judged by its cover? Many would argue that it cannot. This well-known saying serves as a warning against prejudice and an encouragement to look beyond appearances. However, it raises an interesting question: what if there is some truth to it? The choice of a book cover—its colors, composition, imagery, and typography—may contain elements that subtly influence perception.

This project aims to explore whether a book's cover has a measurable impact on readers' judgments using data analysis. The Amazon Books Review dataset from Kaggle will be used, containing hundreds of thousands of books along with their descriptions and reviews.

The objective is to develop neural network models capable of predicting whether a book has received predominantly negative (1-2 stars), neutral (3 stars), or positive (4-5 stars) reviews based solely on its cover image.

The process begins with data acquisition, exploration, target variable creation, and data cleaning. Subsequently, the input and target data will be prepared, including image processing to ensure compatibility with neural networks. The dataset will then be split into training and validation sets.

The model training phase will include a brief theoretical overview of convolutional neural networks (CNNs) and the development of models, starting from simpler architectures and progressing to more complex ones.

Finally, model performance will be evaluated, potential overfitting issues will be addressed, and optimization techniques such as hyperparameter tuning will be applied. The conclusion will summarize the findings and suggest possible improvements for future work.

1 Dataset Description

The dataset used for this project, *Amazon Books Review*, is publicly available on Kaggle and was uploaded by the user Mohamed Bekheet. This dataset consists of two files that provide comprehensive information on books and user reviews.

1.1 Books_rating.csv

This file contains feedback from approximately 3 million users on 212,404 unique books. The data is sourced from the Amazon Review Dataset, which includes product reviews and metadata spanning from May 1996 to July 2014. The file includes the following attributes:

- **id**: Unique identifier for the book
- **Title**: Title of the book
- **Price**: Price of the book
- **User_id**: Unique identifier of the user who rated the book
- **profileName**: Name of the user who rated the book
- **review/helpfulness**: Helpfulness rating of the review (e.g., 2/3)
- **review/score**: Rating given to the book (0 to 5)
- **review/time**: Timestamp of when the review was submitted
- **review/summary**: Summary of the review
- **review/text**: Full text of the review

1.2 books_data.csv

This file provides detailed information about the 212,404 unique books rated in the first file. The data was collected using the Google Books API to enhance book metadata. The attributes in this file include:

- **Title**: Title of the book
- **Describe**: Description of the book
- **authors**: Name(s) of the book's author(s)
- **image**: URL of the book cover
- **previewLink**: Link to access a preview of the book on Google Books
- **publisher**: Name of the publisher
- **publishedDate**: Date of publication

- **infoLink**: Link to additional information about the book on Google Books
- **categories**: Genre(s) of the book
- **ratingsCount**: Average rating count for the book

The features of interest in this project are the book title and the corresponding average review rating and image url. To construct the dataset, a merge will be performed based on the title, which serves as the linking variable between the two datasets. However, before merging, data will be aggregated to calculate the average rating for each book, considering only those with at least 10 reviews. This threshold is chosen to minimize bias from a small number of users while also preventing excessive data loss, as many books have only a few reviews.

2 Dataset Preparation

2.1 Filtering, Merging and Cleaning

The first step in the analysis involves filtering books that have received more than 10 reviews in the initial dataset. The threshold variable ensures that only titles appearing at least 10 times in the dataset are retained. The review scores are then aggregated by calculating their average. The resulting dataset, `avg_scores`, consists of two columns: **Title** and **avg_score**, representing the book title and its average user rating, respectively.

Next, the dataset is merged with the other dataset containing links to book covers, which will serve as the sole input for the model.

Finally, a new column, **Rating**, is created as the target variable. Books with an average rating below 2.8 are classified as Low, those between 2.8 and 3.8 as Medium, and the remaining as High.

After merging and removing null values, the final dataset consists of 36,050 books. Analyzing the distribution of the target variable reveals a strong imbalance, with the High category accounting for approximately 81% of all reviews. This imbalance poses challenges, particularly in the model's ability to distinguish between classes, especially for the Low category, which represents only about 2% of the total data.

This skewed distribution may be attributed to the fact that books receiving a high number of reviews tend to be either popular or well-liked by readers. It is more common to find a large volume of positive reviews for widely known books than for those that were less well-received.

Data augmentation techniques could be applied to increase the dataset size and enhance the model's ability to differentiate between classes. However, for this project, only the available original data will be used, with efforts focused on improving model performance despite the dataset imbalance.

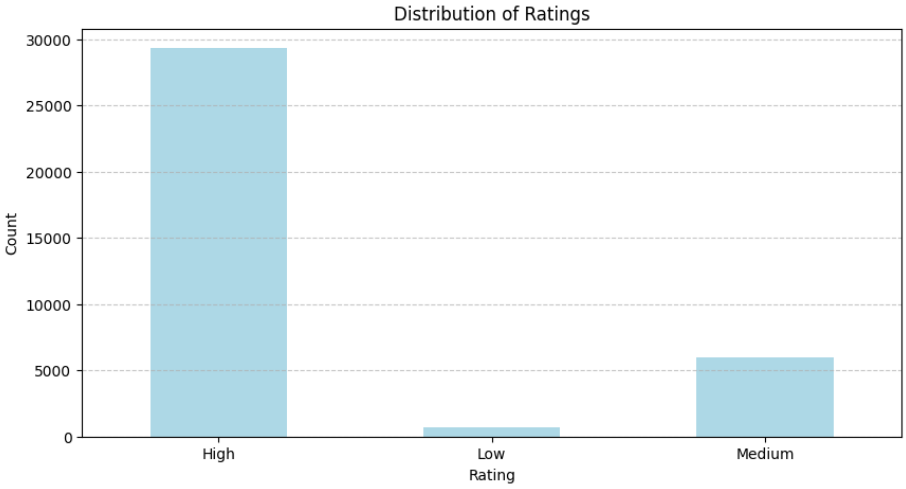

Figure 1: Target Variable Distribution.

2.2 Downloading Cover Images

The next step involves downloading the cover images, as the dataset only provides the links for retrieval. For this reason, I have created a `download_image` function, which is designed to retrieve and save images from a specified URL while handling potential errors. It begins by sending an HTTP request using `requests.get`, with a timeout of 10 seconds to prevent excessive waiting times. To ensure the request is successful, `raise_for_status()` is included, which raises an exception if the response contains an HTTP error. Once the image data is received, it is processed using `BytesIO`, allowing the response content to be treated as a file-like object. The image is then opened with the `Image.open` method from the `PIL` library, ensuring it is correctly interpreted before being saved to the specified filename using the `save` method. To handle possible failures, such as network errors, invalid URLs, or issues with the image format, the entire process is enclosed in a `try-except` block. If an exception occurs, an error message is printed, indicating the problematic URL and the nature of the issue.

To download the images, a loop was implemented using `tqdm` to iterate over the rows of the DataFrame `df`, ensuring progress tracking during execution. The `iterrows()` method was used to access each row individually, retrieving the image URL from the `"image"` column. For each image, a filename was dynamically generated in the format `"covers/{i}.jpg"`, where `{i}` corresponds to the index of the row, ensuring a unique identifier for each file. The images were then saved locally in the `"covers/"` directory. The `download_image` function was called for each URL, handling the retrieval and saving process while managing potential errors.

Finally, a new column `image_path` is created, where each entry corresponds to an image filename based on the DataFrame index with a `.jpg` extension. It then updates the column by prepending the directory path stored in `data_dir` using `os.path.join()`, ensuring that each file path correctly points to its location in the local system.

2.3 Preparing Training and Validation Datasets

At this stage, all necessary components are available to create the training and validation datasets.

First, the target labels are mapped from categorical ratings to numerical values. The ratings, such as `Low`, `Medium`, and `High`, are assigned corresponding integer values, such as 0, 1, and 2, respectively. This transformation is necessary because machine learning algorithms typically require numerical labels rather than categorical ones.

Next, the labels are processed using one-hot encoding. Each label is transformed into a vector where each element corresponds to a potential class. The element that represents the class is set to 1, while all other elements are set to 0. For example, a label of `Medium`, which has been mapped to 1, will be represented as $[0, 1, 0]$ in a three-class classification scenario.

This process ensures that the labels are in a format suitable for training neural network models, which often require one-hot encoded labels for proper training and validation.

The next step involves extracting the image paths from the DataFrame into an array using the expression `df['image_path'].values`. This array contains the file paths for each image that will be used in the training and validation process.

A function is then defined to load and preprocess the images. The function takes two inputs: the image path and its corresponding label. The image is first read using `tf.io.read_file`, which loads the image data from the file system. After that, the image is decoded using `tf.image.decode_jpeg`, specifying that the image should have three color channels (RGB).

Next, the image is resized to a fixed size of 224×224 pixels using `tf.image.resize`. This resizing ensures that all images have the same dimensions, which is crucial for feeding them into neural networks that require a consistent input shape.

The pixel values of the image are then normalized by dividing them by 255.0, ensuring that the pixel values are in the range $[0, 1]$.

Finally, the function returns the preprocessed image and its corresponding label, making them ready for input into a machine learning model during training or validation.

2.4 Creating the Training/Validation Split

In order to create the training and validation split, a TensorFlow dataset is created using `tf.data.Dataset.from_tensor_slices`, where the image paths

and labels are passed as input. This method constructs a dataset by pairing each image path with its corresponding label.

The dataset is then processed using the `map` function, which applies the `load_image` function to each element in the dataset. This is done in parallel using `num_parallel_calls=tf.data.experimental.AUTOTUNE`, allowing TensorFlow to automatically optimize the number of threads used for loading and preprocessing the data, improving efficiency.

Next, the dataset is shuffled using `dataset.shuffle(len(image_paths))`, which randomizes the order of the data to ensure that the model does not learn any unintended patterns from the order in which the data is presented. This step is important for preventing overfitting and ensuring that the training process is robust.

The dataset is then batched into groups of 32 samples using `dataset.batch(batch)`, to reduce computation time and memory usage. Additionally, `dataset.prefetch(tf.data.experimental.AUTOTUNE)` is used to prefetch data, meaning that while the model is training on one batch, the next batch is already being prepared in the background, further speeding up the process.

The dataset is split into training and validation sets. The training set is created by taking the first 70% of the dataset, determined by `train_size = int(0.7 * len(dataset))`. The remaining 30% of the dataset is allocated to the validation set, which is obtained by skipping the first `train_size` elements and taking the next `val_size` elements.

It is now possible to proceed with constructing the architecture for training a neural network model.

3 Convolutional Neural Networks

Before proceeding with the practical implementation of the model, a theoretical background on convolutional neural networks is provided. These information are elaborated from the notes of the course `Deep Learning for Computer Vision` from Stanford, available at the following link: `https://cs231n.github.io/convolutional-networks/#layersizepat`.

Convolutional Neural Networks (CNNs) are a type of neural network specifically designed for processing data with a grid-like structure, such as images. Unlike traditional neural networks, where each neuron in a layer is fully connected to all neurons in the previous layer, CNNs take advantage of the input's spatial structure and impose a more efficient architectural design.

In particular, CNN layers are organized in three dimensions: width, height, and depth. This structure allows each neuron to connect only to a small region of the previous layer, known as the *receptive field*, rather than being fully connected to all neurons. This local connectivity and the parameter-sharing scheme (where multiple neurons within the same "depth slice" share the same weights) make CNNs highly scalable for large images while significantly reducing the number of parameters, thus mitigating the risk of overfitting.

CNN architectures typically consist of a sequence of different types of layers,

each serving a specific function. The three main types of layers used to construct CNNs are:

- **Convolutional Layer (CONV)**: The core computational block, applying a set of learnable filters to the input to extract local features.
- **Pooling Layer (POOL)**: Progressively reduces the spatial dimensions of the representation, decreasing the number of parameters and computational complexity while also helping to control overfitting.
- **Fully-Connected Layer (FC)**: Similar to those in traditional neural networks, it is typically placed at the end of the architecture to produce the final class scores, where each neuron is fully connected to all activations in the previous volume.

Additionally, after a CONV layer, a non-linear activation function such as ReLU is often applied to introduce non-linearity and improve learning efficiency.

4 Neural Network Architectures Used in Training

In the next section, the architectures of the three developed neural network models will be described. The implementation starts with a simple model containing a single convolutional layer and progressively increases in complexity by adding more layers, ultimately leading to deep learning models.

4.1 First Model: Basic CNN

The first model is a basic convolutional neural network with a minimal architecture. It consists of a single `Conv2D` layer with 8 filters of size $(3, 3)$, followed by a `MaxPooling2D` layer to reduce spatial dimensions. The output is then flattened and passed through a `Dense` layer with 16 neurons and ReLU activation before reaching the final output layer with 3 neurons and softmax activation for multi-class classification. This model serves as a foundational approach with a lightweight structure.

4.2 Second Model: Deeper CNN

The second model introduces a deeper architecture with three convolutional layers. The first `Conv2D` layer has 16 filters, followed by a `MaxPooling2D` layer. Two additional convolutional layers with 32 and 16 filters, respectively, are added, each followed by max pooling to progressively reduce the spatial dimensions. After flattening, the model includes a fully connected `Dense` layer with 256 neurons before the softmax output layer. This architecture captures more complex spatial patterns compared to the first model.

4.3 Third Model: Deep CNN

The third model further extends the depth by adding an additional `Conv2D` layer, making it a deeper CNN. It follows a similar structure as the second model but incorporates four convolutional layers instead of three, enhancing its ability to learn hierarchical features. Each convolutional layer is followed by max pooling, and the fully connected part remains the same, with a `Dense` layer of 256 neurons before the softmax output. This model represents the most complex architecture, leveraging multiple convolutional layers to extract detailed spatial representations from the input data.

These three models demonstrate a progressive approach, from a simple CNN to deeper architectures capable of more sophisticated feature extraction. Note that all architectures were trained using early stopping, which is is a regularization technique used in machine learning to prevent overfitting by stopping training when the model's performance on a validation set starts to degrade. In the following sections, the models' performance and comparative analysis will be explored.

5 Training Results

Below are the results of the initial training phase.

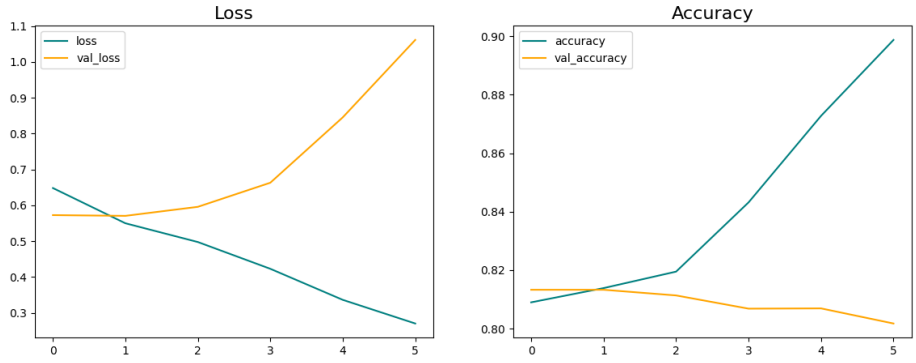

Figure 2: Basic CNN.

Given that this is a simple neural network with only one convolutional layer, high performance was not expected. As shown in the graph, the model does not perform well and quickly tends toward overfitting. This is evident from the sharp decline in the blue line (representing training loss) and the rapid increase in the orange line (representing validation loss). Similarly, the accuracy on the training set rises sharply, while that on the validation set gradually declines. These results highlight the need to increase the architectural complexity of the model to improve its generalization ability.

Here the results of the second architecture:

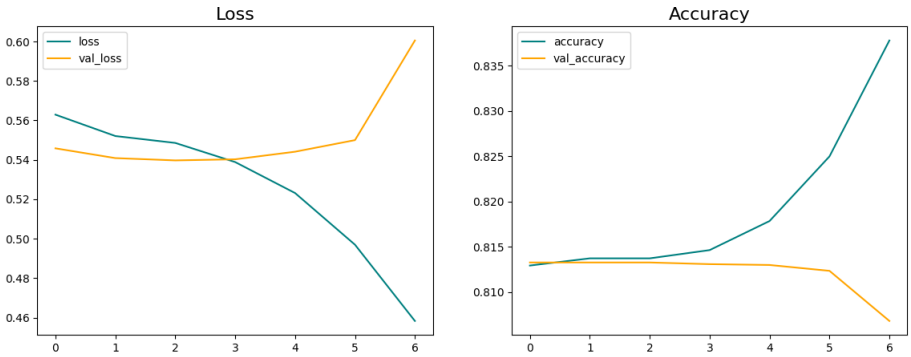

Figure 3: First Deep Learning Architecture.

Unlike the first trained model, adding a few additional convolutional layers leads, as expected, to a slight improvement in performance. However, this model also quickly exhibits overfitting and demonstrates limited generalization ability.

Let us proceed with the final implementation:

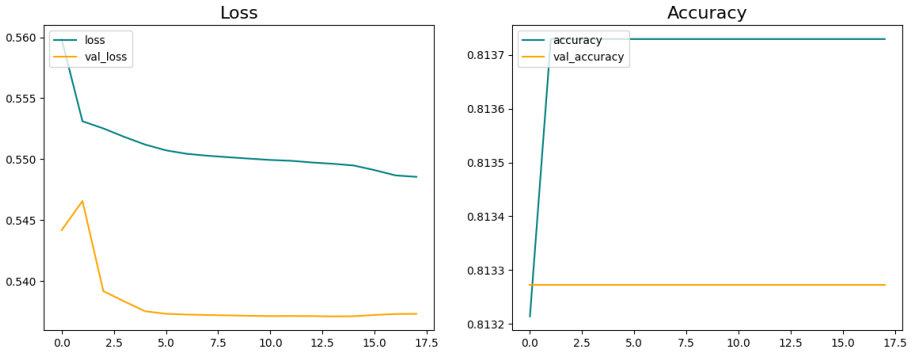

Figure 4: Second Deep Learning Architecture.

The final model was built with the most complex architecture. Its results appear significantly more stable, with a sharp initial drop in validation accuracy that then stabilizes, while training accuracy remains relatively constant. This architecture seems to provide more robust performance, with a reduced tendency

toward overfitting.
So far, dropout has not been included in any of the previous architectures. Dropout is a regularization technique used in neural networks to prevent overfitting. During training, it randomly deactivates a percentage of neurons in each layer, preventing the network from relying too heavily on specific connections. This enhances the model's ability to generalize to new data. The next step involves incorporating dropout layers into the last architecture to assess whether it improves performance and enhances generalization.

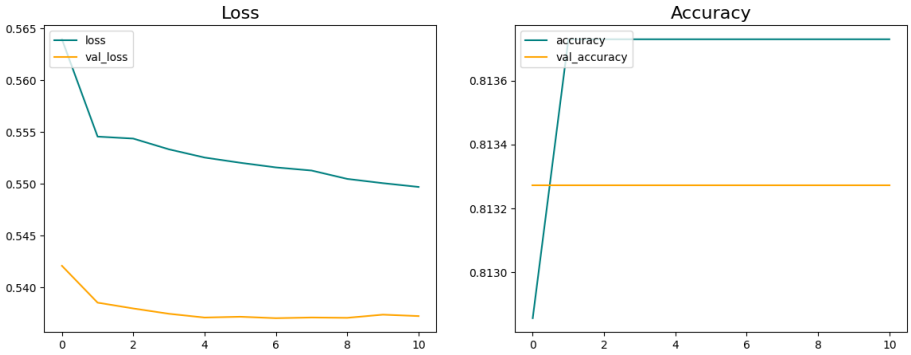

Figure 5: Addition of a Dropout Layer.

The addition of dropout layers further stabilizes the training process. The validation loss no longer exhibits the sharp peak observed in the previous training and aligns more closely with the training loss. The accuracy follows a similar trend to the prior training, but in this case, the gap between the training and validation sets is narrower, indicating that the model is now better at generalizing than before.

6 Model Evaluation and Conclusions

Examining the results obtained across different models, it is evident that increasing architectural complexity led to progressive improvements in training stability.

The first model exhibited a high tendency toward overfitting, with a validation loss of 0.5695 and an accuracy of 0.8132. The gap between training and validation results indicated limited generalization capability.

The second model incorporated additional convolutional layers, slightly reducing the loss (0.5397) and stabilizing accuracy (0.8133), signaling a minor improvement.

The third model continued this positive trend, further lowering the loss

(0.5371) while maintaining the same accuracy. However, a noticeable gap between training and validation results persisted, indicating residual overfitting.

Finally, the introduction of dropout in the last model resulted in the most balanced configuration. The loss decreased to 0.5370, accuracy remained steady at 0.8133, and training appeared more stable, with a reduced discrepancy between training and validation performance.

In conclusion, incorporating dropout proved to be an effective regularization strategy, mitigating overfitting and enhancing the model's generalization ability. However, since accuracy remained unchanged, further improvements might be achieved through techniques such as hyperparameter tuning or expanding the dataset with more diverse examples.

7 Hyperparameter Tuning

The final step of the project involves utilizing various hyperparameter values to optimize the model's performance. Hyperparameter tuning in machine learning is the process of finding the best values for hyperparameters, which control the model's behavior but are not learned from the data. The goal is to improve the model's generalization ability and maximize its performance.

In this case, four different training sessions will be performed, testing all possible combinations between two learning rates and two batch sizes. For the learning rate, a standard value of 0.01 and a very low value of 0.0001 will be used. Lower learning rates can slow down learning but make it more stable, reducing the risk of getting stuck in local minima. Regarding batch size, training will be performed using both the previously used value (32) and a new value (16). Larger batch sizes reduce the variance in gradient estimation, speeding up convergence but can lead to less generalizable solutions. Smaller batch sizes increase the stochasticity of updates, favoring better generalization, but with noisier and less stable training.

Afterward, the performance of these different models will be analyzed and compared with previous results, followed by a final comparison. The architecture used will be the one from the last training, which includes the dropout layer.

Below is a summary of the training combinations:

- **Training #1:** Learning rate: 0.01, batch size: 32
- **Training #2:** Learning rate: 0.0001, batch size: 32
- **Training #3:** Learning rate: 0.01, batch size: 16
- **Training #4:** Learning rate: 0.0001, batch size: 16

7.1 Results

The analysis of results obtained with different combinations of learning rate and batch size highlights minimal differences in final performance but provides useful insights into the stability of training.

- **Model #1 (lr = 0.01, batch size = 32):** Achieved a loss of 0.5374 and accuracy of 0.8133.
- **Model #2 (lr = 0.0001, batch size = 32):** With a loss of 0.5391 and accuracy of 0.8133, a slight degradation compared to the first model was observed, suggesting that a very low learning rate may have made optimization slower and less effective.
- **Model #3 (lr = 0.01, batch size = 16):** Achieved a loss of 0.5374 and accuracy of 0.8134. Using a smaller batch size seems to have slightly improved accuracy compared to the first model, suggesting that the network benefited from more frequent weight updates.
- **Model #4 (lr = 0.0001, batch size = 16):** With a loss of 0.5390 and accuracy of 0.8134, the results are similar to Model #2, confirming that a very low learning rate did not lead to significant advantages.

The results suggest that a higher learning rate (0.01) generally led to a slightly lower loss, indicating more effective optimization with a value that is not too low. Additionally, a smaller batch size (16) seems to have slightly improved accuracy, likely because it allowed for more frequent weight updates.

Overall, Model #3 (lr = 0.01, batch size = 16) appears to be the most performant configuration among those tested, with the best combination of loss and accuracy. However, the differences between the models remain minimal, suggesting that further improvements could come from other tuning strategies or a deeper exploration of hyperparameters.

8 Final Evaluation and Improvement Strategies

How can we truly assess if the model is performing well? An accuracy of 81% seems high, which could lead us to believe that the model is performing perfectly. However, as mentioned earlier, the dataset exhibits highly imbalanced values, meaning that a high accuracy is exactly what we would expect. To evaluate the model more effectively, it is useful to compare it with a baseline percentage to determine whether it is truly discriminating between classes or merely exploiting the data imbalance to achieve high accuracy without learning to distinguish the classes meaningfully.

A good starting point is to compare the model's accuracy with that of a naive classifier, which always predicts the most frequent class. If our model achieves an accuracy similar to this baseline, it indicates that it is not truly learning meaningful patterns in the data but is simply leveraging the class distribution to attain high accuracy.

The fact that the target value is identical to the model's performance suggests that it may not have actually learned complex patterns in the data and is merely replicating an existing distribution present in the dataset. This reinforces the need to further explore optimization strategies, not only at the model

level but also in the management of the training data. A crucial aspect in this regard is the quality and quantity of the data used: refining the criteria for selecting books based on the minimum number of reviews might represent a further step toward improving dataset diversity and, consequently, the model's ability to generalize better to new examples. Therefore, it may be worth trying to modify the minimum review threshold required to include a book in the training process. Currently, books with fewer than 10 reviews have been excluded to reduce the risk of skewing the model with unrepresentative ratings. However, lowering this threshold (e.g., to 5 reviews) could increase the number of data points available for training, thereby enhancing the dataset's diversity and robustness.

A larger dataset could improve the model's generalizability by including more books with diverse visual characteristics. Moreover, this approach could make the dataset more balanced, reducing any distortions resulting from excluding less popular works. On the other hand, this modification involves a trade-off, as reducing the minimum number of reviews per book might increase the risk of distorted judgments since reviews with few ratings tend to be less representative of the overall consensus.

However, this solution did not yield the expected results. I implemented a function, `distribution_by_threshold`, that allows us to analyze how the class distribution varies when the minimum review threshold is changed. The results show that lowering the threshold does not significantly reduce the dataset's imbalance.

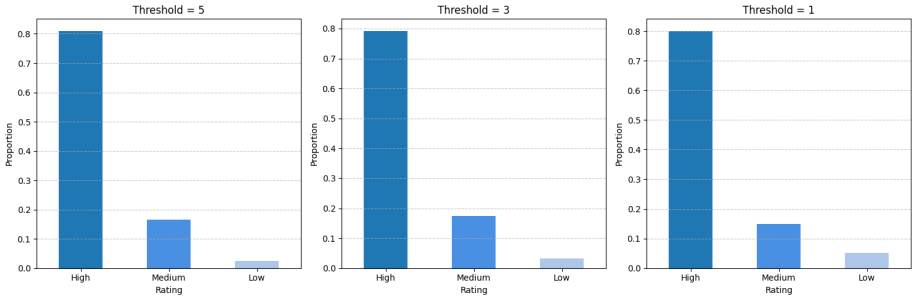

Figure 6: Target Variable Distribution with Different Thresholds.

For example, with a threshold of 5 reviews, the dominant class `High` still represents 81% of the data, while the `Medium` and `Low` classes remain in a marked minority, at 16.5% and 2.4%, respectively. Reducing the threshold further to 3 reviews results in a slight decrease in the proportion of `High` to 79.2%, but the gap with the other classes remains substantial. Finally, lowering the threshold to 1 review, the `High` class still makes up 80% of the dataset, while the other two categories show only marginal variations.

These results indicate that increasing the number of books in the dataset

does not necessarily lead to a more balanced class distribution. In fact, the dominance of the `High` class remains virtually unchanged, suggesting that the imbalance issue is not solely caused by excluding books with fewer reviews, but rather by an intrinsic characteristic of the dataset itself. Therefore, to improve the model, it may be necessary to adopt different strategies.

To further improve the model's performance, two complementary strategies can be adopted: data augmentation and the integration of new features. Data augmentation could be applied to the images, generating artificial variations through transformations that might help the model generalize better, reducing the risk of overfitting, and improving its ability to recognize relevant patterns.

Another interesting solution involves incorporating textual information as an additional feature. The original dataset contains a column with book descriptions, which could provide further context useful for the model. To implement this solution, a multimodal architecture could be adopted, combining both images and text. In practice, the text could be processed through an embedding model to convert it into a numerical representation. This embedding could then be concatenated with the features extracted from the convolutional network processing the images. The resulting model would be a neural network with two separate inputs—one for images and one for text—that merge into a final dense layer for classification. This approach could enhance the model's predictive ability by leveraging both visual information and