Sentiment Analysis of Airbnb Reviews

Project Report

Samuel Castillo

July 22, 2024

Table of Contents

Supplemental Contents

Section 1: Introduction

As the age of remote work and global connectedness develops in the wake of the COVID pandemic, many Americans are now free to travel more frequently and explore new locales with relative ease. Vacations are no longer limited to week-long summer trips – many remote workers spend multiple weeks traveling throughout the year simply because their work accommodates it. Even university students are more available to travel in the post-COVID online availability of their coursework. In just a few short years since the pandemic, the "nomadic" lifestyle has given rise to several unique developments in social and economic infrastructures. For instance, the rise in short-term travel has had an unsurprising yet tremendous impact on the short-term property rental industry through platforms like Airbnb. According to Statista, Airbnb received over 448 million bookings in 2023 alone, a near 100% increase since 2018[1].

Naturally, Airbnb's growing popularity has led to a stark increase in short-term real estate investment. Buyers are consistently looking for properties in tourist-filled areas to add to their Airbnb rental portfolios. It's a hit-or-miss game, though – not every area is suitable for a profitable investment. To address this uncertainty, platforms like airDNA release market-value index scores for short-term rentals based on propriety metrics. As helpful as these scores are, the model characteristics and input data are hidden to maintain secrecy, which makes it difficult to know why certain investments may be more sensible. Moreover, airDNA has a paywall, which may deter new or frugal investors.

1.1 Project Scope

This project is an integral step in a larger effort to produce a custom predictive model to inform short-term real estate investments similar to airDNA. However, unlike airDNA, the model characteristics and input data will be transparently accessible to the public through an open-source Github repository.

The final model will account for several metrics related to both Airbnb popularity and general property value. The first step towards this holistic approach rests on the Airbnb-specific data taken directly from the platform's website – particularly in the context of user reviews. A thorough gauge of reviews can reveal quite a bit about the historical appreciation of properties in a region, making a review analysis a suitable contribution to the short-term investments model.

A comprehensive review analysis involves several Natural Language Processing (NLP) techniques that engineer meaningful numeric features based on unstructured text. This project entails the analysis of user-sentiment to determine whether a review is positive, negative, or neutral in tone – this feature informs the likelihood of whether users will re-book their rental experience or recommend it to a friend. Ultimately, a sentiment-analysis

[1] "Airbnb - Statistics & Facts," Statista, accessed July 22, 2024, https://www.statista.com/topics/2273/airbnb/#topicOverview.

feature will allow the final model to inform investment decisions according to the flexibility of historical unstructured user input.

1.2 Project Details and Results

The sentiment analysis model is an optimized rendition of Hugging Face's DistilBERT NLP neural network. The final model trained and tested on 500k Airbnb reviews sampled from thousands of listings in 894 cities across the US. The network predicted sentiment classes for the unseen test data with 98.97% accuracy, 98.58% precision, and 99.01% recall. Overall, the network performed extremely well as a candidate for the sentiment analysis component of the short-term investments model, making this project a success.

Section 2: Extracting Airbnb Review Data

There are two primary means of accessing and collecting Airbnb data: the developer's API and the end-user website. Though the API is generally much faster and more efficient for collecting large volumes of data compared to the website, it has a paywall. Since this project began with an established budget of $0, the API was out of scope simply by virtue of costing anything. So, all review data was gleaned directly from the end-user website through a custom-built web-scraper.

2.1 Web-Scraping: BeautifulSoup vs Selenium

Web-scraping is a powerful means of extracting data from public websites in an automated fashion. Generally, web-scraping programs are designed to access specific URLs, find elements of interest, and return relevant data without requiring a user-managed browser instance. In other words, web-scrapers are automated crawlers that find and collect web-based content without user oversight. For this project, manually copy-pasting reviews from thousands of Airbnb listings would have taken at least one year of continuous work, so web-scraping was the only viable option to gather the necessary data.

The most common web-scraping libraries available in Python are BeautifulSoup and Selenium. BeautifulSoup is comparatively simple to use and allows easy access to website metadata as well as common HTML elements. However, BeautifulSoup is built primarily for static content – it does not handle web navigation and interaction easily. Selenium, on the other hand, is more complex under the hood but allows dynamic web navigation. Since this project required review collection from multiple listings for each city, Selenium was a more suitable choice given its dynamic capabilities.

2.2 Web-Scraping Procedure

Selenium operates using a web driver – essentially, the library instantiates its own browser within the system and executes commands in succession, allowing dynamic interaction and navigation. The procedure of gathering Airbnb reviews for this project used the following framework:

1. Identify the generic HTML class for an Airbnb listing. This required a detailed inspection of the HTML structure of the Airbnb website.
2. Instantiate a web driver and direct it to a specific city's Airbnb landing page. These URLs each bear the format: "www.Airbnb.com/s/{city}/homes". Landing pages contain the first twenty available rental listings with the option to navigate to further pages for more listings.
3. Collect each listing on the first five pages (approximately 100 listings total) based on the HTML class identified above. Each item in the resulting list is a "clickable" item that navigates to a specific rental listing page.
4. Navigate to each listing page collected in the previous step, record the URL, insert the string "/reviews" at the appropriate position, and store the new URL in a list. The new URL navigates to a page of available reviews for that listing.
5. Instantiate a new driver to navigate to each review URL in the list from the previous step. Scrape up to 200 of the most recent reviews from each listing until 1500 reviews have been collected.
6. The final product to be stored: 1500 reviews taken from 18 to 100 listings representing rental properties in a single city.

This procedure was repeated for a list of 894 different cities, each of which was chosen due to its presence in the Zillow housing research dataset that forms the other substantial portion of the short-term real estate investments model. After the reduction of duplicates and nulls, the number of reviews totaled just under one million.

2.3 Edge-Cases and Error Handling

One of the main challenges in web-scraping is dealing with the inconsistent nature of web content. Pages may or may not load correctly, and any given page can easily be missing a crucial element that all pages like it have in common. Consequently, the web-scraper developed for this project required approximately six major version overhauls to cover edge cases. Ultimately, each block of code within the final script was fitted with try-except blocks to pass over failed attempts without shutting down the program.

2.4 Cloud Computing in Google Cloud Platform

At peak performance, the scraper was capable of gathering approximately 1500 reviews (one city's worth) in thirty minutes on a standard machine. With 894 cities to cover, the total runtime would have amounted to approximately 13 days of 100% CPU use on a single unit. To avoid this extended runtime, the job was parallelized across six *n1-standard1* Linux virtual machines in a cluster through the Google Cloud Platform. The entire task was complete within 3 days.

```python
while reviews_captured < max_reviews:
    reviews = driver.find_elements(By.CLASS_NAME, review_class)
    current_review_count = len(reviews)

    if current_review_count <= reviews_captured:
        break

    for review in reviews[reviews_captured:]:
        if reviews_captured >= max_reviews:
            break

        driver.execute_script("arguments[0].scrollIntoView();", review)

        try:
            name = review.find_element(By.CLASS_NAME, "hpipapi").text
            rating_info = review.find_element(By.CLASS_NAME, "s78n3tv").text
            review_text = review.find_element(By.CLASS_NAME, "r1bctolv").text

            review_data.append({
                "name": name,
                "rating_info": rating_info,
                "review_text": review_text
            })

            reviews_captured += 1

        except Exception as e:
            print(f'Could not extract review data: {e}')
            continue
```

Figure 3. Web-Scraper Review Capture. A small portion of the web-scraping program captures relevant review info based on HTML structures. Reviews are saved in lists of dictionaries.

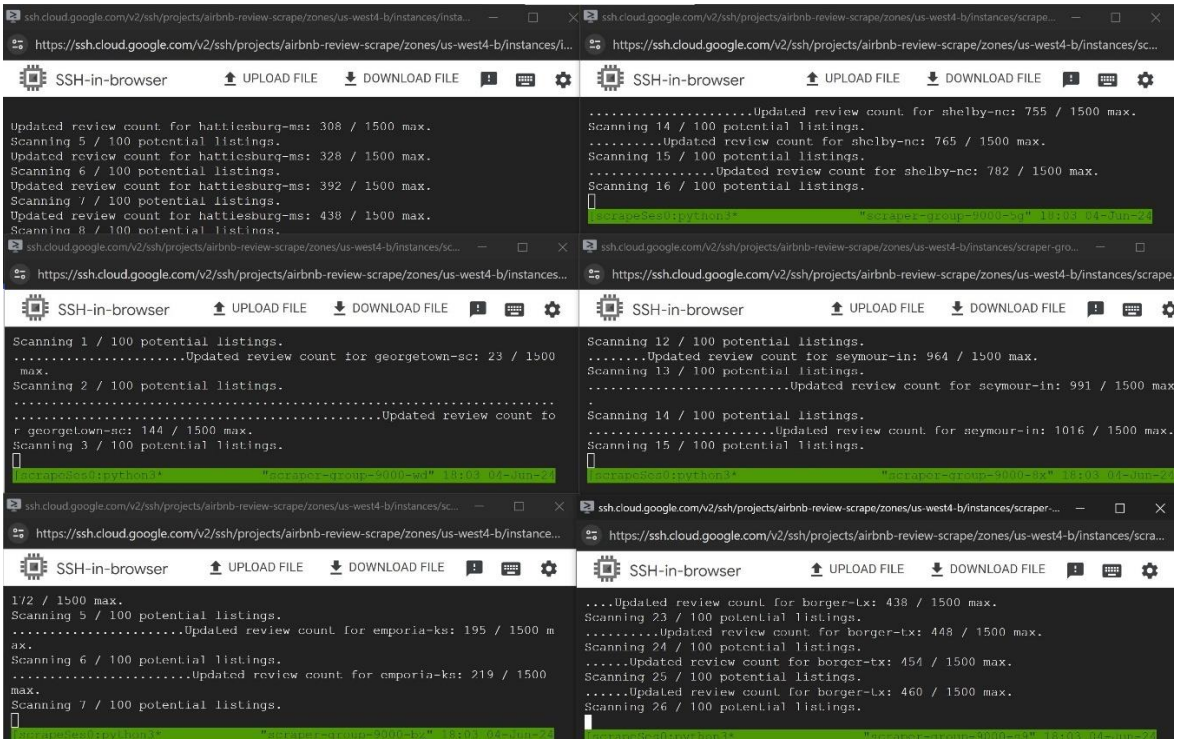

Figure 4. Linux VM Cluster. Six virtual machines operate in parallel to capture 894 total cities' worth of reviews.

To accommodate the virtual Linux system, the web-scraping script was adjusted to run using Firefox (rather than the original Chrome) without a graphic interface (i.e., "headless"). Each job ran in a tmux detached session to allow easy tracking, and all adjustments and actions were drafted in VSCode and executed at the Linux command line.

Reviews from each of the six machines were collected and loaded into the same Google Cloud Bucket, from which they were ultimately downloaded to the local machine for processing and analysis.

Section 3: Preprocessing and Exploring Review Data

The scraper produced separate csv files for each city's reviews to accommodate concatenation with city-specific property value data for the second step of the project. However, to build the sentiment analysis model, all reviews needed to be concatenated into one document to allow more effective training and evaluation. But first, the data required preprocessing. The raw review data arrived in the following format:

	name	rating_info	review_text
0	Sarah	Rating, 5 stars\n,\n·\n3 days ago\n,\n·\nStaye...	Stephen's place was amazing. The interior was ...
1	Kris	Rating, 5 stars\n,\n·\n1 week ago\n,\n·\nStaye...	Stephen's historic home exudes timeless elegan...
2	David	Rating, 5 stars\n,\n·\n2 weeks ago\n,\n·\nStay...	Very cozy place, I would definitely stay again.
3	Luca	Rating, 5 stars\n,\n·\n1 week ago\n,\n·\nStaye...	2nd time staying there overnight! Really nice ...
4	Kristen	Rating, 5 stars\n,\n·\n2 weeks ago\n,\n·\nStay...	My dogs and I loved our stay at the cottage ho...

Figure 5. Raw Review Data. The "rating_info" column is messy and inconsistently formatted.

Naturally, this was not conducive to a productive analysis. The 'rating_info' column was delimited by new line markers, the timestamp was inconsistently formatted, and the 'name' feature was relatively useless. After cleansing, review data looked like the following:

	date	rating	note	review_text
0	2024-06-01	5	Stayed a few nights	Stephen's place was amazing. The interior was ...
1	2024-06-01	5	Stayed a few nights	Stephen's historic home exudes timeless elegan...
2	2024-06-01	5	Stayed a few nights	Very cozy place, I would definitely stay again.
3	2024-06-01	5	Stayed with a pet	2nd time staying there overnight! Really nice ...
4	2024-06-01	5	Stayed with a pet	My dogs and I loved our stay at the cottage ho...

Figure 6. Processed Review Data. The "rating" column provided labels and the "review_text" provided features for the supervised ML sentiment analysis task.

3.1 Class Imbalance

From here, review data was much easier to explore and analyze. A basic exploratory investigation was conducted in a Tableau Dashboard for ease of use and interaction. The most noteworthy observation was the drastic class imbalance present in the review set – there were over 846k 5-star reviews and only 848 2-star reviews.

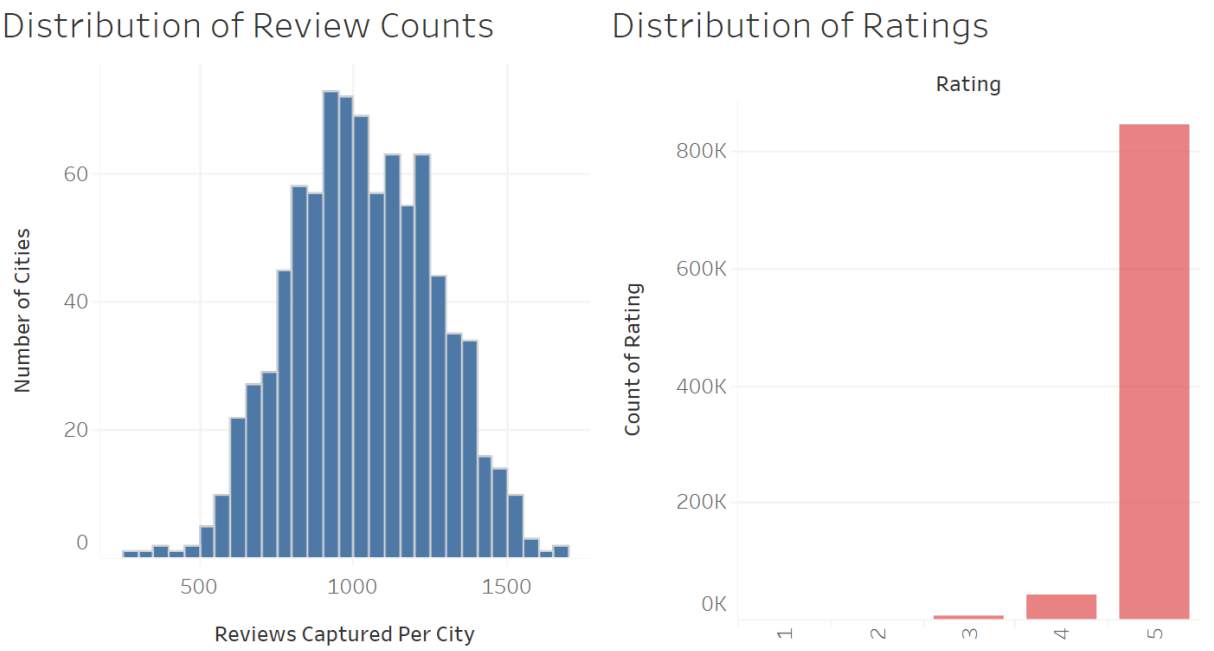

Figure 7. Tableau Dashboard of Review Metrics. After addressing nulls and duplicates, most cities offered around 1000 reviews. The rating imbalance was quite drastic with an 800k difference in quantity between majority and minority classes.

The imbalance could have arisen due to a number of factors. At the root, the scraper can be adjusted in future renditions to capture an even amount of each rating type. Regardless, an imbalance of this magnitude could easily lead to an improperly tuned model that appears accurate simply by chance. Thus, exploratory analysis revealed that certain measures must be taken to avoid the pitfalls of class imbalance – namely, these measures involved grouping classes and over/under-sampling accordingly.

Further exploration revealed certain patterns within the text which would form the basis of the NLP analysis. Word clouds demonstrate the prevalence of words and phrases according to specific conditions – though the final model operates on a much more complex level, word frequency analysis forms an intuitive foundation for the mathematical principles behind natural language processing.

Figure 8. Word Cloud of Low Ratings. Certain themes display more prevalence: need, dirty, uncomfortable, etc.

Figure 9. Word Cloud of High Ratings. Opposite to low ratings, high ratings display prevalence of words such as beautiful, comfortable, amazing, great location, etc.

After consideration of a few exploratory measures, the review data was ready for modeling.

Section 4: The Sentiment Analysis Model

Sentiment analysis models function as most classic supervised machine learning methods: the model refines an algorithm that receives multiple features as inputs in order to predict the value of a single target variable. After several iterations of training data, the model learns the relationship between predictors and targets to develop a generalizable predictive algorithm, which can be evaluated on unseen data to determine validity. In the case of this

project, the inputs were the user reviews, and the target variable was the rating class – negative (1-2 stars), neutral (3 stars), or positive (4-5 stars).

Unlike many machine learning tasks, however, sentiment analysis relies on millions of inputs and engineered interactions between inputs to make a valid prediction of user sentiment. Consequently, traditional models such as Random Forest Classifiers or even Gradient Boosting Machines are far too cumbersome and generally unable to handle the computational power required to conduct thorough sentiment analysis tasks. Instead, NLP objectives are most commonly executed via deep learning frameworks, i.e., neural networks.

4.1 Neural Networks

Neural networks receive their namesake from the complex connectivity patterns observed in the human brain – biological neurons are renowned for receiving thousands of inputs and transmitting information at unmatched speeds, and neural networks are a computational effort to emulate this wonder of nature to process data.

The fundamental structure of neural networks is relatively simple: input features feed information into a hidden layer of neurons, and those neurons subsequently transform and pass information through any number of hidden layers that each perform their own transformations. The final hidden layer outputs to the predicted values, classes, or probabilistic representations of the target variable. The network then evaluates the loss of predictions versus the actual labels, backpropagates through each hidden layer of neurons, and adjusts the numeric transformations between each layer to reduce loss in the next iteration. This process occurs until the model achieves a satisfactory loss or performs well when working with unseen test data.

4.2 Weights, Biases, and Learning Rate

The "transformations" that occur between hidden layers are vectorized combinations of weights and biases. Weights describe the direction and magnitude by which an input is altered from a source neuron to a destination neuron. Biases are additive measures in the destination neuron that affect that neuron's tendency to "fire" – each neuron has a threshold of required activity according to its activation function, and biases nudge the input one way or another to alter the likelihood of meeting that threshold. In other words, weights and biases are adjustable parameters between and within neurons that affect the strength of the original feature input as it works its way through the network.

Initially, the values of weights and biases are entirely randomized as the network is "unfamiliar" with the relationship between inputs and labels. As the model continues to pass the data over multiple epochs, the network applies gradient descent to a loss function to find values of weights and biases that minimize loss (i.e., the adjusted difference between true values and predicted values). The rate at which it adjusts parameters according to the gradient descent is the learning rate – a higher learning rate can result in faster convergence, but it can also cause the descent to pass over minima and fail to converge at

all. Ultimately, neural networks require extensive optimization of the learning rate and other hyperparameters to produce models that generalize well to unseen data while remaining within computational limits and expenses.

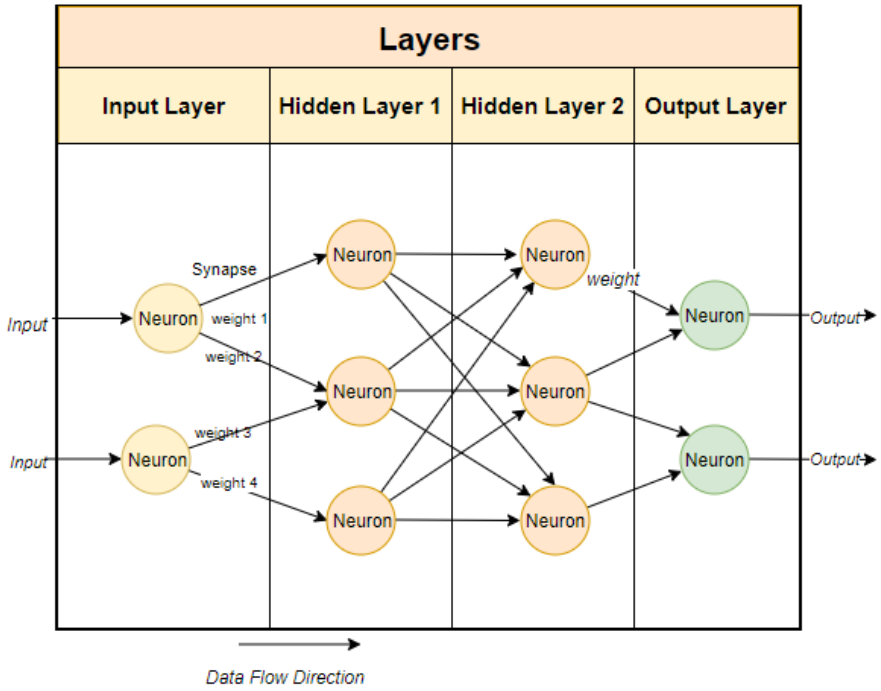

Figure 10. Basic Structure of a Neural Network. Neural networks possess interconnected hidden layers between input and output layers.

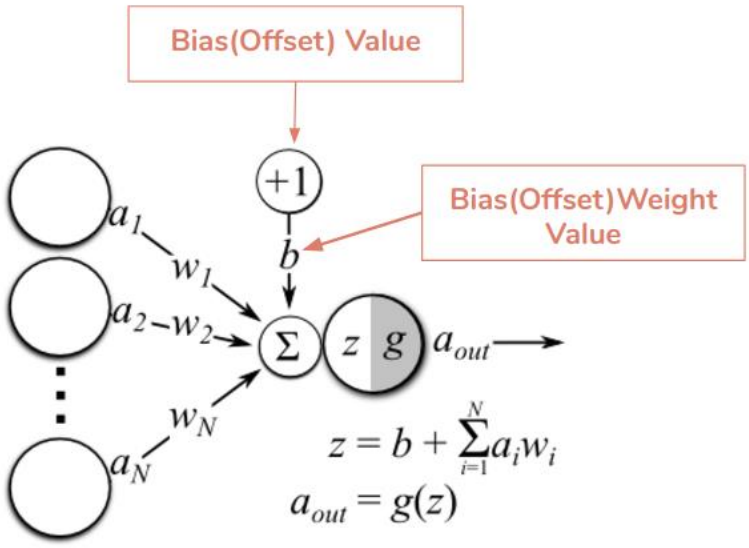

Figure 11. Weights and Biases Represented. Each destination neuron receives weight-adjusted inputs from source neurons, the summation of which is offset by a unique bias and passed through an activation function and an additional weight before transmitting signal to the next neuron.

The modular structure of layers, weights, and biases allows neural networks to detect interactions between features as well as interactions between those interactions – the depth at which these models can discover high-dimensional patterns is limited only by computational power. This makes neural networks ideal for language processing tasks, which require a thorough understanding of how words interact in sequence given a certain context.

4.3 Transfer Learning: DistilBERT Model

Creating neural networks from scratch to accomplish complex NLP tasks requires a great deal of computational resources, often more than personal machines can handle. Transfer learning provides a practical alternative by utilizing pre-trained models that have been built and trained on powerful machines. This approach allows application of these models for specific tasks beyond their original context. This project utilized transfer learning techniques to employ the DistilBERT sentiment analysis model from the Hugging Face transformers library. DistilBERT is a lightweight rendition of the original BERT (Bidirectional Encoding Representations from Transformers). DistilBert is pre-loaded with a vocabulary of 30,000 words and designed to handle sentences and paragraphs efficiently while optimizing computational resources.

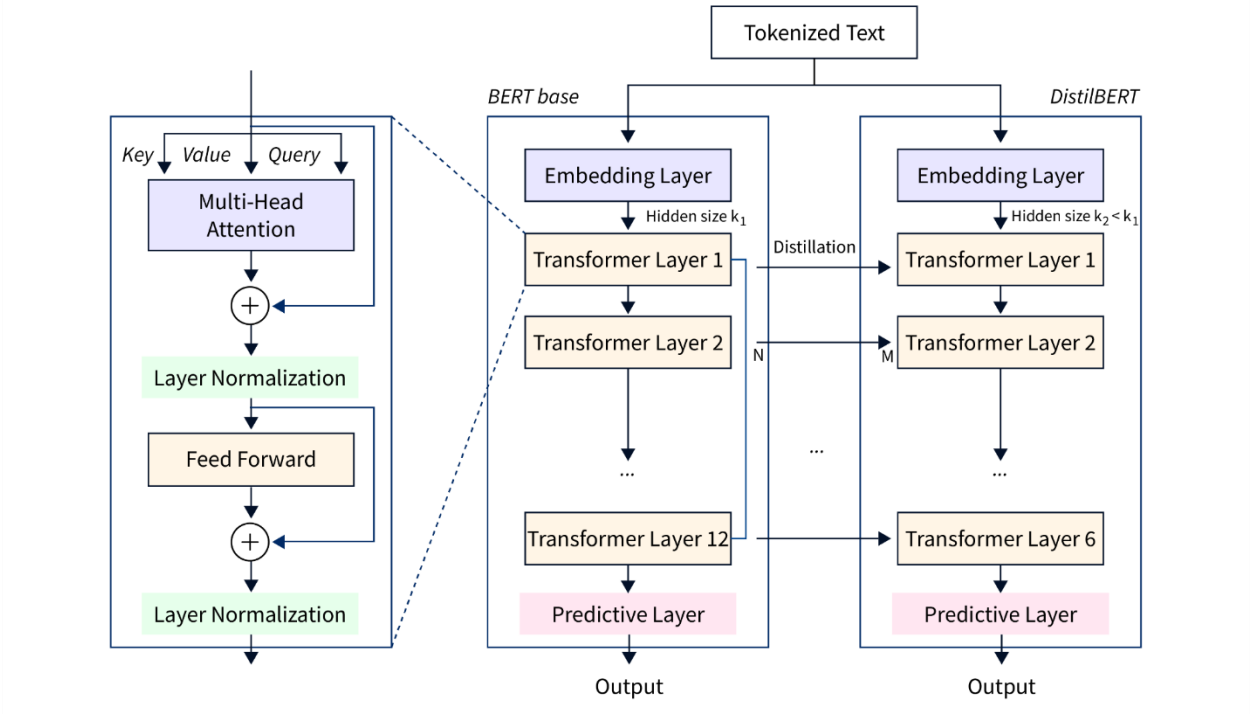

Figure 12. DistilBERT Basic Architecture. The lightweight rendition of BERT possess embedding, transformer, and predictive layers to make classifications from tokenized text.

4.4 DistilBERT Mechanism

The DistilBERT Sentiment Analysis model performed a complex sequence of steps to arrive at a final prediction of sentiment class. Broken down, the process was as follows:

1. Tokenization & Padding

The DistilBERT Tokenizer transformed the raw input string to a list of words and subwords. For the sake of consistency, each list was trimmed or padded to a length of 128 tokens.

Example: The sentence "This place was amazing!" might be tokenized into ['This', 'place', 'was', 'amazing', '!']. Since there are fewer than 128 tokens in this case, the sequence would be padded with special padding tokens to reach the threshold.

2. Input ID Conversion

Each term in DistilBERT's extensive vocabulary was associated with a numeric label out of the box. Following tokenization, tokens were converted to these proprietary numeric IDs to maintain consistency across samples and allow interaction with the DistilBERT framework. For ease of interpretability, DistilBERT automatically added tokens to the beginning and end of each sequence for later processing ([CLS] and [SEP] respectively).

Example: After tokenization, "This place was amazing!" might be converted to IDs [101, 2023, 2780, 2001, 6203, 999, 102] where 101 and 102 represent the [CLS] and [SEP] tokens, respectively.

3. Attention Mask

Attention is a binary indicator of whether the model should "attend to" a token or not. Tokens that represent padding are typically masked as 0, whereas all other tokens are masked as 1. The attention mask ensured that padding tokens did not interfere with the model's processing of the sequence.

Example: For the sequence with padding, the attention mask might look like [1, 1, 1, 1, 1, 0, 0, …, 0] where 1s correspond to real tokens and 0s correspond to padding.

4. Positional Encoding

The network did not inherently maintain the order of tokens based on the original input. Thus, an extra step was required to create vectorized positional encodings of the same dimension as the size of hidden layers as determined by the model configuration – in this case, encodings represented 768 dimensions. This positional information allowed the model to account for the order of tokens in the original sequence as well as relative positioning between tokens; in other words, position involved a token's place in a sentence as well as its relationship to every other word in that sentence. Ultimately, this positional vector would be added to the embedding layer created in the next step.

Example: The token 'This' might receive a positional encoding vector that tells the model it is the first token in the sequence. This positional encoding is later combined with the token embedding.

5. Embedding Layer

The embedding layer converted each token ID from the raw input string into a dense vector. Each token ID was mapped to a fixed-size vector of 768 dimensions using an embedding matrix. Embedding matrices are learnable parameters that adjust during training to better capture the semantic properties of the tokens.

- Token Embeddings: Each token ID was mapped to a dense vector using the embedding matrix of size V * d, where V is the vocabulary size and d is the embedding dimension (e.g., 768 for DistilBERT). These vectors captured the semantic meaning of the tokens.
- Addition of Positional Encodings: The positional encodings calculated in step 4 were added to the token embeddings to incorporate positional information. This combined representation was then passed to the transformer layers.

Example: The token 'This' is converted to a 768-dimensional vector using the embedding matrix. Positional information for its position in the sequence (e.g., the 1st token) is added to this vector.

6. Transformer Layers

DistilBERT contains six hidden layers by default, though the number of layers was subject to change during hyperparameter optimization to find the ideal value for this task. Regardless, each layer consisted of two major sequential steps: Multi-Head Self Attention Mechanism and Position-wise Fully Connected Feed-Forward Network.

- Multi-Head Self Attention allowed the model to process each token in relation to every other token in the input. Essentially, each input vector was updated with normalized attention scores that blended information from all tokens into the representation of each token. This process took place in parallel with multiple heads to capture various aspects of token relationship simultaneously.
- The final output from the self-attention mechanism was further processed by the Position-wise Fully Connected Feed Forward Network. Each token representation passed through two linear transformations with a rectified linear activation function to allow the model to understand more complex patterns.

Input values underwent this two-step process in each hidden layer; the output of one layer served as the input to the next. Ultimately, the final transformer layer returned a set of richly contextualized token representations that captured intricate relationships between tokens and complex sequential patterns.

Example: In self-attention, the model might determine that ‘amazing’ and ‘place’ are related, which adjusts their representations. Each layer refines these representations based on token relationships.

7. Output Representation

In this sentiment analysis task, the final representation of the first token in the input sequence (CLS) served as a contextual summary of the entire string. The final hidden state of CLS was represented as a vector with 768 dimensions, identical in shape to how it entered the transformer layers after Step 5.

Example: The [CLS] token’s final vector captures the overall sentiment of "This place was amazing!" as a single 768-dimensional vector.

8. Classification Layer

The CLS vector then passed through another layer to provide a set of logits. Since this task involved classification of three labels (negative, neutral, positive), the classification layer returned three logits for each input string.

Example: For the input "This place was amazing!", the classification layer might output logits [2.3, -0.5, 1.7] corresponding to positive, negative, and neutral classes, respectively.

9. Softmax Layer

The final step in the network was a softmax activation function that converted each of the three logits to a probabilities for each class.

Example: The logits [2.3, -0.5, 1.7] might be transformed into probabilities [0.75, 0.05, 0.20], indicating a high probability for the positive class.

10. Prediction

Ultimately, the final label prediction was the class with the highest probability. The model compared these final predictions to the actual labels to calculate loss and update each learnable parameter throughout the network accordingly.

Example: Given the probabilities [0.75, 0.05, 0.20], the model predicts the sentiment as positive.

4.5 Implementing DistilBERT: PyTorch

Building and training the DistilBERT neural network required framework that could accommodate multiple high-dimensional vectorized operations. The PyTorch library served this need well through tensor datasets and dynamic graphs – tensors are vector-like data structures optimized for neural network processes, and dynamic graphs are computational “blueprints” that allowed flexible implementation and customization of the model configuration. PyTorch also integrates seamlessly with HuggingFace’s Transformers library,

which made it simple to define, customize, and train the DistilBERT model through PyTorch's framework.

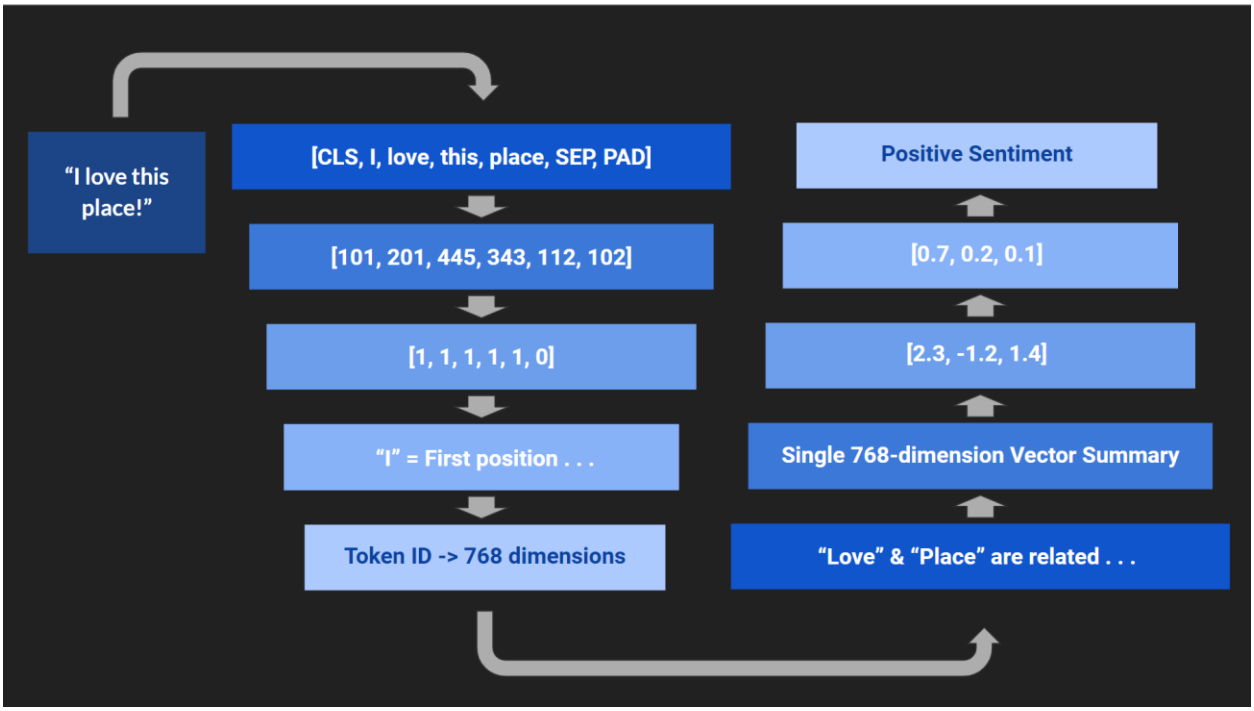

Figure 13. Graphic Representation of DistilBERT Mechanism. The phrase "I love this place!" is likely to be assigned a class of positive sentiment through the complex ten-step DistilBERT mechanism.

```
# Dataset class for handling encodings and labels
class SentimentDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: val[idx] for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx], dtype=torch.long)
        return item

    def __len__(self):
        return len(self.labels)
```

Figure 14. PyTorch Dataset Definition. In order to pass data through the DistilBERT architecture, it must pass as tensors, which are vector-like data-structures known as Datasets in the PyTorch ecosystem.

Section 5: Optimizing Configuration and Hyperparameters

Once built, the DistilBERT model performed relatively poorly for sentiment classification "out of the box." Accuracy was below 40%, and it seemed obvious that the model was making random guesses without learning based on the dataset. Similar to traditional machine learning approaches, neural networks require hyperparameter optimization to establish the most effective configurations for the task at hand.

Unlike traditional machine learning models, however, the hyperparameter search space for neural networks is much, much larger. The primary difference lies in the weights and biases – each weight and bias is revised multiple times throughout the training process, adding hundreds of thousands of adjustments to the optimization. Moreover, the configuration of neural networks is generally more extensive than regular machine learning models, widening the search space even further.

The weights and biases within the transformer layers of the DistilBERT network were adjusted automatically after each pass-through according to the gradient descent operations of the loss function discussed previously. Thus, the hyperparameters that required intentional optimization were integral to the model's configuration. These variables are included in the following table:

Table 1. Hyperparameters Included in Optimization Search Space

Hyperparameter	Description	Effect of Increasing	Effect of Decreasing
Number of Hidden Layers	This is the number of layers in the transformer part of the model. Each layer consists of attention and feed-forward components that help the model understand complex patterns in the data.	Can capture more complex patterns but may lead to overfitting and longer training times.	Reduces model complexity, which may prevent overfitting but could also result in underfitting if the model becomes too simple.
Dropout Rate	Dropout is a regularization technique where a fraction of the neurons is randomly turned off during training to prevent overfitting.	Helps prevent overfitting by making the model more robust, but too high a rate can lead to underfitting.	Allows the model to learn more from the data but may increase the risk of overfitting.
Attention Dropout	This is the dropout rate applied specifically to the attention weights within the transformer layers.	Similar to general dropout, helps prevent overfitting in the attention mechanism, but too much can reduce the model's ability to focus on relevant parts of the input.	Improves the model's ability to learn from the attention weights but might increase overfitting.
Learning Rate	The learning rate determines how much to adjust the model's weights with respect to the loss gradient during	Speeds up training and helps the model converge faster, but too high a rate can cause the	Provides a more stable and precise convergence but can make training very slow and

	training.	model to overshoot optimal solutions and result in instability.	potentially get stuck in local minima.
Weight Decay	Weight decay is a regularization technique that penalizes large weights in the model, effectively preventing overfitting by adding a term to the loss function.	Encourages smaller weights, thus reducing overfitting, but too much weight decay can lead to underfitting.	Allows the model to learn more freely but may result in overfitting if the model becomes too complex.
Warmup Steps	Warmup steps are the initial steps during which the learning rate gradually increases to its peak value before potentially decaying.	Provides a smoother and more stable training start, preventing drastic updates early on. However, too many warmup steps can slow down the overall training process.	Allows the learning rate to reach its peak value faster, which can accelerate learning, but might cause instability if the model updates too aggressively early in training.
Number of Epochs	The number of epochs is the number of complete passes through the training dataset.	Allows the model more opportunities to learn from the data, potentially improving performance, but too many epochs can lead to overfitting.	Reduces the risk of overfitting and saves training time but might result in underfitting if the model doesn't learn enough from the data.

5.1 Optuna: Bayesian Optimization

The search space for these seven hyperparameters was quite extensive, as seen below:

```
def objective(trial):
    # Define hyperparameters to tune
    learning_rate = trial.suggest_float('learning_rate', 1e-5, 5e-5, log=True)
    num_train_epochs = trial.suggest_int('num_train_epochs', 2, 4, step=1)
    dropout_rate = trial.suggest_float('dropout_rate', 0.05, 0.75)
    attention_dropout = trial.suggest_float('attention_dropout', 0.05, 0.55)
    num_layers = trial.suggest_int('num_layers', 4 ,6, step=1)
    weight_decay = trial.suggest_float('weight_decay', 1e-6, 1e-3, log=True)
    warmup_steps = trial.suggest_int('warmup_steps', 100, 1000, step=100)
```

Figure 15. Hyperparameter Search Space. The search space involved float and integer spans over 7 different hyperparameters.

Many traditional machine learning methods rely on a grid search or randomized search to find the most suitable combination of hyperparameters, but that approach would have been inefficient and likely ineffective due to the expanse of the search space. Instead,

optimization was conducted using the Optuna library, which relied on multiple trials of Bayesian optimization – essentially, the optimizer updated its preferences for certain dimensions of the search space based on the most promising values. Thus, each trial resulted in a further optimized model rather than a random variation of hyperparameters.

Optimization was conducted using 20,000 reviews from the original dataset, which represented a small portion of the 900k+ reviews available. This saved several days of processing time while still allowing the model to learn based on a variety of data. Majority classes were under-sampled and minority classes were over-sampled to ensure a valid distribution and accommodate train-test stratification.

Due to computational demand, Optuna trials were conducted in Google Cloud Platform on a Linux Virtual Machine similar to the web-scraping task. However, instead of using an n1-standard1 single core machine with 10GB of disk space, running the DistilBERT optimization required a c2 standard8 machine with 8 cores and 30GB of available disk space. Trials were designed to run all 8 cores in parallel to optimize computational efficiency, and the entire process took approximately 4 days.

5.2 Optuna Trials

The optimizer ran five total trials with various hyperparameter configurations. The configurations and final results are in the table below:

Table 2. Optuna Trials Hyperparameters and Results

Hyperparameter	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
Learning Rate	2.84629e-05	3.34058e-05	4.56523e-05	1.60789e-05	3.15227e-05
# Epochs	3	2	2	3	4
Dropout Rate	0.07384	0.58472	0.25767	0.47595	0.22544
Attention Drop.	0.37841	0.17470	0.05682	0.16366	0.30899
# Layers	6	4	6	6	4
Weight Decay	1.58100e-05	3.19483e-06	3.95673e-06	4.04230e-05	3.94177e-04
Warmup Steps	800	900	600	600	700
Eval Loss	0.06823	0.28544	0.09716	0.12317	0.05976
Eval Accuracy	0.98264	0.88503	0.97530	0.96345	0.98514
Eval Precision	0.97711	0.89100	0.96519	0.95708	0.97977
Eval Recall	0.98275	0.82549	0.97919	0.95727	0.98575
Eval F1	0.97986	0.84186	0.97164	0.95717	0.98268

According to the data, Trial 5 is the most successful with eval metrics that surpass those of every other trial. However, the configurations that Trial 5 ran involved only 4 hidden transformer layers compared to DistilBERT's standard 6 layers. While the 4 layers performed well on a smaller dataset of 20,000 reviews, the ultra-light rendition of the model did not breed confidence in generalizing well to patterns hidden in hundreds of thousands of reviews. If 4 layers were enough to maintain power over larger datasets,

Hugging Face probably would have chosen 4 instead of 6. Thus, the model that moved to the final stage was the second-best performing model for the sake of deeper complexity: the model generated in Trial 1. Trial 1's eval metrics are step-for-step with Trial 5, and the added hidden layers provide greater promise of excellent performance on a larger scale. A further look at metrics and results for the Optuna trials can be seen in the following figures.

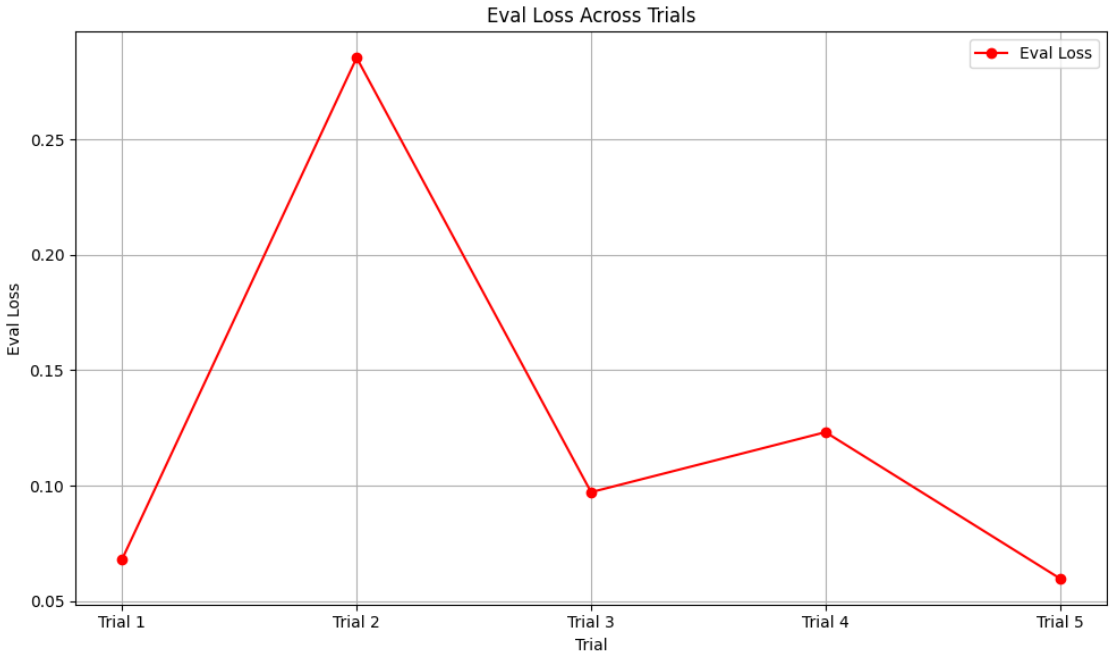

Figure 14. Evaluation Loss Across Trials. The loss increases sharply after the first trial as the model attempts to explore the search space, but it sharply returns to 0.10 as the Bayesian optimizer corrects hyperparameters based on previous knowledge.

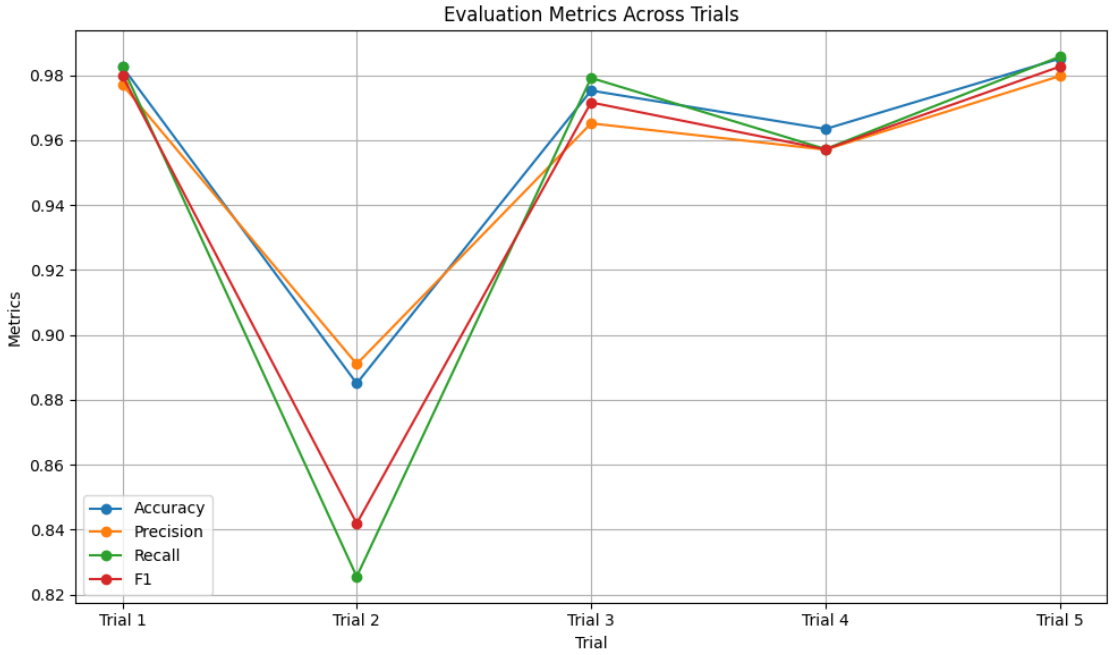

Figure 15. Evaluation Metrics Across Trials. The four evaluation metrics follow a pattern similar to evaluation loss.

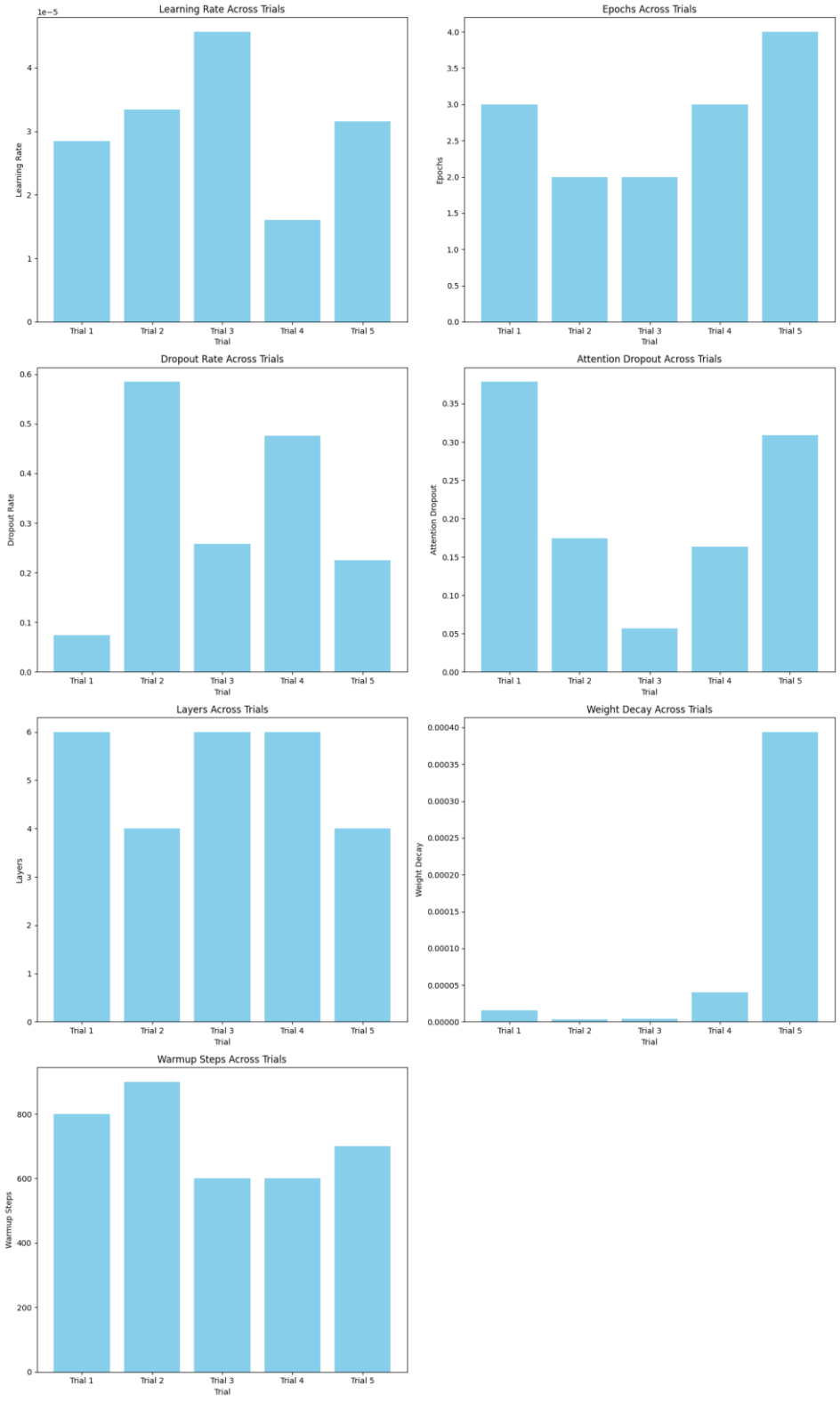

Figure 16. Hyperparameters Across Trials. The Bayesian Optimization process is evident through the progression of trials for each hyperparameter. Each subsequent trial is an attempt to correct errors borne from the configuration of previous trials.

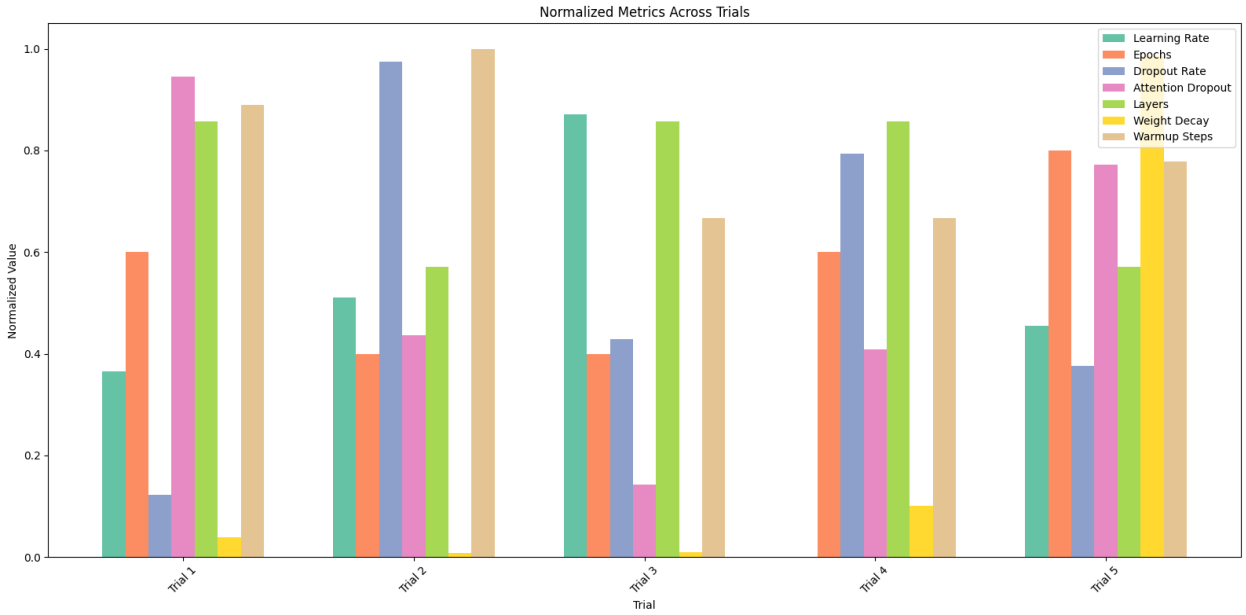

Figure 16. Normalized Metrics Across Trials. Each hyperparameter is shown on a normalized scale for each trial. In this manner, relative patterns of Bayesian correction are more easily discernable between hyperparameters.

5.3 The Final Model

The final model with hyperparameters taken from Trial 1 was run on a larger scale to allow the network to learn on a more substantial portion of data. The disk space in the c2-standard8 machine didn't accommodate running the network with all 900k reviews collected by the web-scraper, but it did allow 500k reviews. The final metrics are below:

Table 3. Final Model Metrics and Results	
Hyperparameter	**Trial 1**
Learning Rate	2.84629e-05
# Epochs	3
Dropout Rate	0.07384
Attention Drop.	0.37841
# Layers	6
Weight Decay	1.58100e-05
Warmup Steps	800
Eval Loss	0.03358
Eval Accuracy	0.98974
Eval Precision	0.98582
Eval Recall	0.99008
Eval F1	0.98791

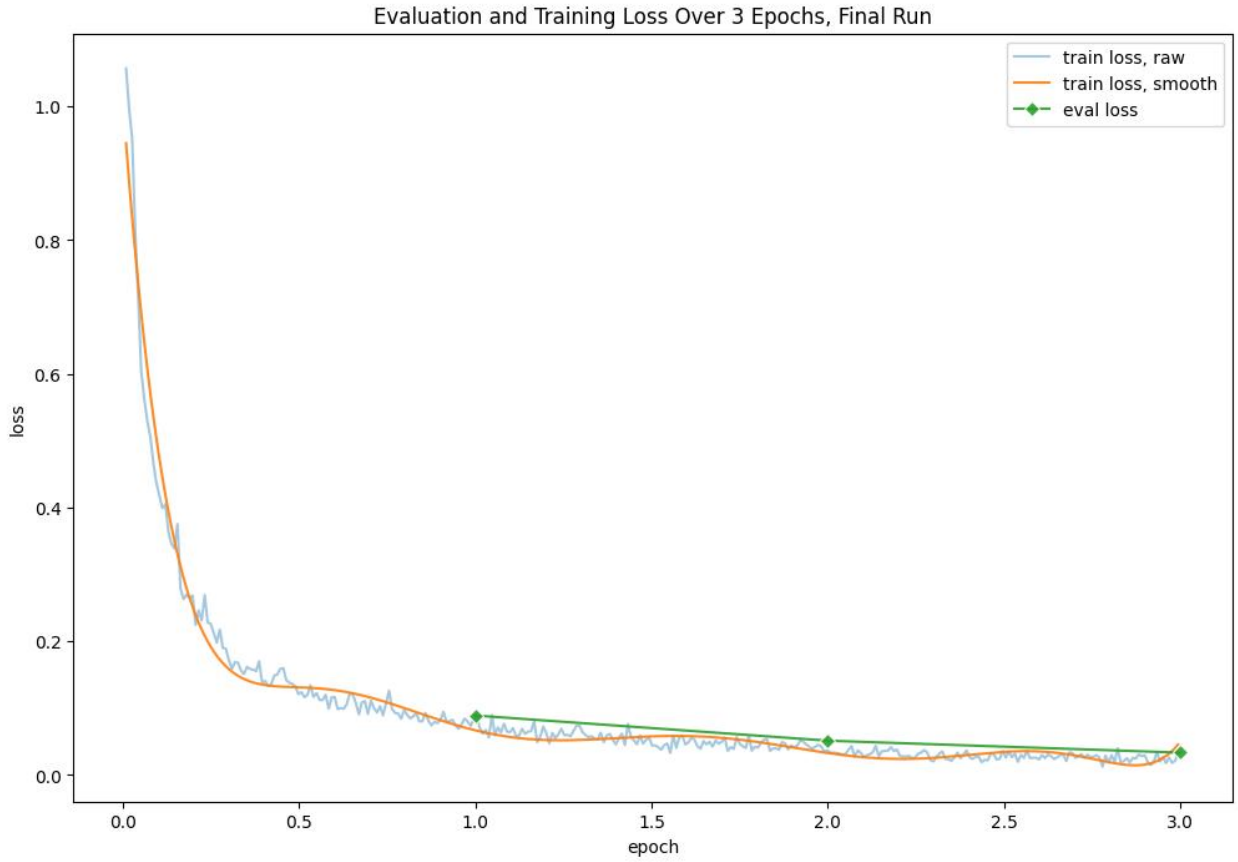

Figure 17. Evaluation Loss for Final Model. The final model produced a final evaluation loss of 0.03 by the third epoch. Loss fell drastically following the first quarter-epoch, which indicates a healthy learning rate and weight decay.

Section 6: Conclusion

Given the model's stellar performance in the final evaluation and predictions on unseen test data, this project is considered a success. The custom DistilBERT network is ready for integration with other algorithms and datasets to build a holistic model for short-term real estate investment.

6.1 Potential Revisions

Given a more extensive timeframe and greater computational resources, this project could undergo multiple revisions to improve the overall approach and quality of the final model. For one, it's possible to engineer more features that lend insight to sentiment based on the data collected from Airbnb. Length of review text, special notes (e.g., "stayed with kids"), geographic indicators, and time-series data can all contribute to the feature pool. Moreover, the classification of labels based on ratings may potentially be more effective if assigned differently – rather than groupings of 1-2, 3, and 4-5 stars, perhaps groupings of 1-3, 4, and 5 stars would have addressed the class imbalance more effectively.

Moreover, a wider hyperparameter search space and greater number of Optuna trials (perhaps 10-20) would ensure a more effective model in future renditions. To this end, employing a more powerful machine (ideally with GPUs) would make a tremendous difference.

Overall, though the model itself would benefit from more engineered features and extensive fine-tuning, it performs extremely well on unseen review data and is suitable for the next stage of the project.

Acknowledgements

Much gratitude and credit are granted to the following for making this project possible:

- Ale Berbesi, Technical Mentor and Project Supervisor
- Victor Frank, Sr. Cloud Architect and Neural Network Consultant
- Rachel Castillo, Wife and Greatest Encourager
- Benedict Castillo, Son and Greatest Motivator