Yelp Rating Predictions

Raguvir Kunani

May 11, 2021

Abstract

I use deep learning approaches to predict the ratings of Yelp reviews given the review text, with the goal of standardizing Yelp ratings across all establishments and users. I also deploy the best model as an API. This project was completed as part of the Spring 2021 iteration of Full Stack Deep Learning at UC Berkeley.

1 Introduction

Currently, there exists a problem where between 2 distinct reviewers' perceptions of a 5-star (or any-star) rating (of a restaurant, for example) may be vastly different. Building a model that attempts to predict ratings from reviews could be useful to Yelp and other business review sites in an effort to standardize how businesses are rated. The above situation could potentially be mitigated by using the output of a model to recommend a rating to the reviewer prior to publishing the review. Based on model performance and business needs, these sites could even enforce ratings to be generated algorithmically. My intention is to create a model that predicts user ratings given text of a review, then deploy the model as a tool for suggesting ratings to users.

2 Data

In principle, the above problem applies to all online ratings services (Amazon reviews, Google reviews, Yelp reviews, etc.). However, to narrow the scope of the project to make it feasible to complete in a few weeks, I focus on Yelp reviews. To this end, I will make use of Yelp's public dataset.

The Yelp dataset consists of 8M+ reviews (in addition to other data such as user data, tip data, etc.) and spans the years 2004-2021. In order to ensure that I am learning the "current" mapping from review text to rating, I only use reviews from 2019-2021 (around 1.7M reviews). An additional motivation for using a subset of the data is compute limitations (I used Colab for model training).

3 Modeling

3.1 Metrics

Before diving into model selection, I thought about what metrics I think are important for the task. Since ultimately the problem is a classification problem, accuracy is an obvious metric of importance. However, since accuracy is easy to game in the face of class imbalance, I inspect the confusion matrices

[1] of the models to get a heuristic measure of precision/recall. Note that there is class imbalance in the dataset (Table 1).

Rating	Percent
5	51.8
4	15.2
3	7.8
2	6.9
1	18.2

Table 1: Distribution of Ratings of Reviews

Additionally, the ordinal nature of the classes lends itself to another metric of importance. When the model is wrong, we would like it to make "small mistakes". Since the classes are ordinal, I use mean absolute error (MAE) as a metric to measure how "off" our model's predictions are.

3.2 Baselines

To give some meaning to the learned model's performance, I compute the above metrics on a few baseline approaches.

3.2.1 Random Predictions

In this baseline approach, each prediction is sampled uniformly from $\{1, 2, 3, 4, 5\}$. Technically, a better random baseline would be to sample from the empirical rating distribution, but since the predictions are random anyway I keep the uniform distribution for simplicity. As expected, we see around 20% accuracy (Figure 1).

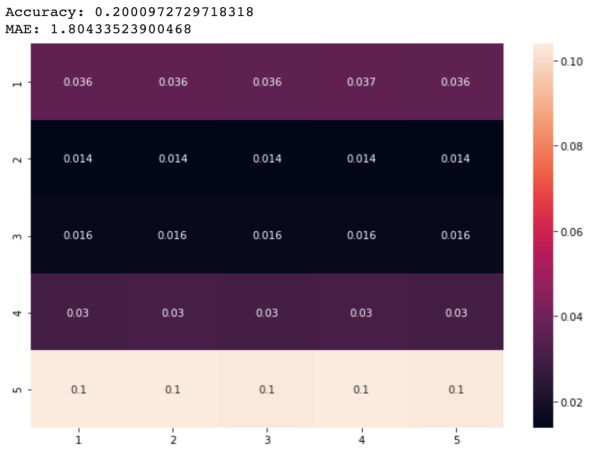


Figure 1: Metrics for Random Prediction Baseline

[1] By definition a confusion matrix C is such that C_{ij} is equal to the number of observations known to be in group i and predicted to be in group j.

3.2.2 Average Restaurant Rating

In this baseline approach, each predicted rating is the average of that establishment's ratings in the dataset (rounded to ensure that the prediction is an integer). This approach unsurprisingly has a much higher accuracy than random guessing, at around 35% (Figure 2).

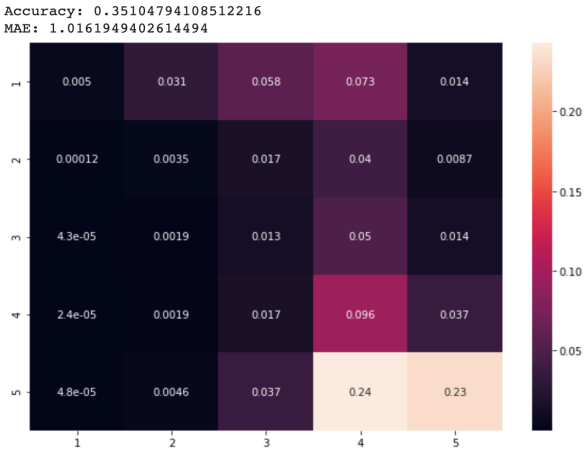


Figure 2: Metrics for Average Establishment Rating Baseline

3.2.3 Average User Rating

In this baseline approach, each predicted rating is the average of that user's ratings in the dataset (rounded to ensure that the prediction is an integer). The accuracy is 37.2% and the MAE is 0.896. See Figure 3 for the confusion matrix

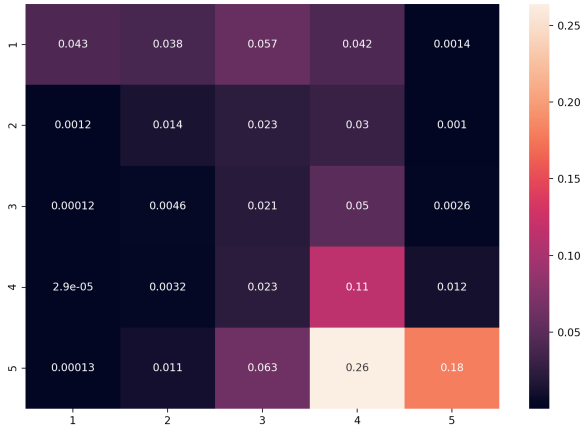


Figure 3: Metrics for Average User Rating Baseline

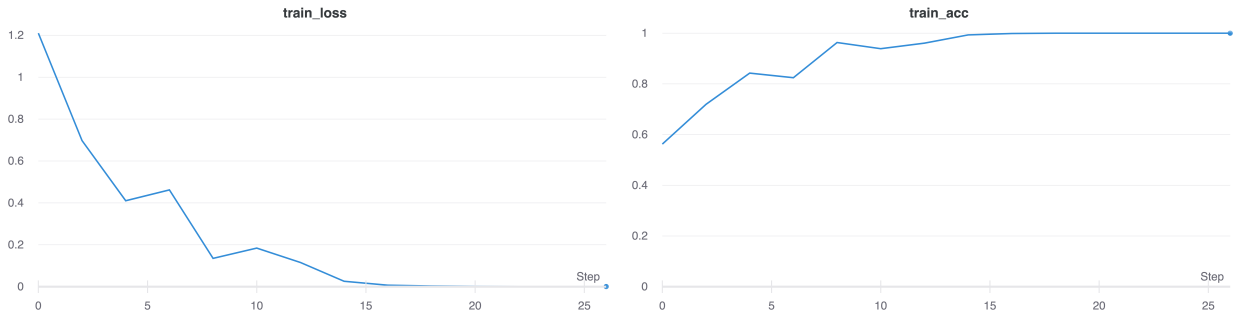
3.3 Deep Learning

3.3.1 Architecture Selection

Since the task is an NLP task, I use a pre-trained language model followed by a fully-connected classification head as my model. Due to compute and memory constraints (both in training and deployment), I use DistilBert from Huggingface Transformers as the pre-trained language model.

3.3.2 Overfit a Small Dataset

To make sure the model has enough expressive capacity to learn the mapping from review text to ratings, I first overfit the model to a set of 1000 reviews. The loss curve steadily decreases to 0 and the accuracy curve increases to 1, so I am confident in the model's expressive capacity.



(a) Train Loss

(b) Train Accuracy

Figure 4: Overfit the model to a set of 1000 reviews

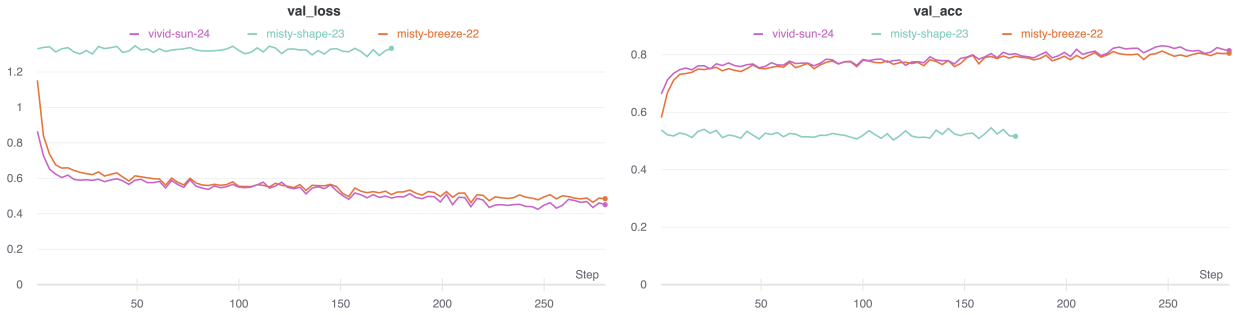
3.3.3 Model Training

In the model training phase, I trained the model on the entire dataset (reserving 10% for validation). Most of the work I did here was setting up experiment tracking with Weights and Biases and experimenting with the optimizer learning rate value and learning rate schedule.

I first did a manual sweep over the learning rates $\{.001, .0001, .00001\}$, with the results shown in Figure 5. Both $.0001$ (vivid-sun-24) and $.00001$ (misty-breeze-22) yielded similar learning curves.

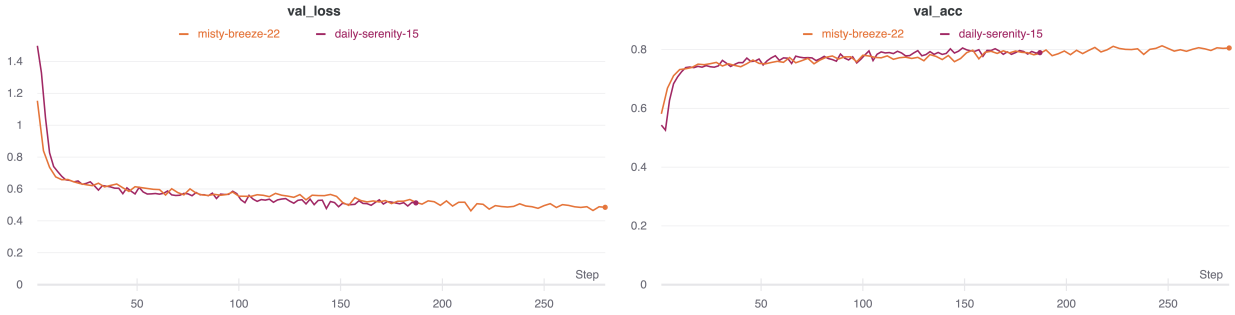
I also experimented a bit with learning rate schedule, but it did not end up affecting the learning curves (Figure 6). In daily-serenity-15, the learning rate is linearly decreased over the course of training. In misty-breeze-22, the learning rate halves when the validation loss plateaus for 10 steps.

At this point, I moved onto deployment with the intention of coming back and doing more model training. However, due to deployment being unexpectedly difficult (discussed in Section 4), I did not have time for other model architecture and training experiments I wanted to do (discussed in Section 5).



(a) Validation Loss (b) Validation Accuracy

Figure 5: Learning Curves for Different Learning Rates



(a) Validation Loss (b) Validation Accuracy

Figure 6: Learning Curves for Different Learning Rate Schedules

Figure 7 shows the metrics of the model I deployed (vivid-sun-24) computed on a validation set of 1680 reviews.

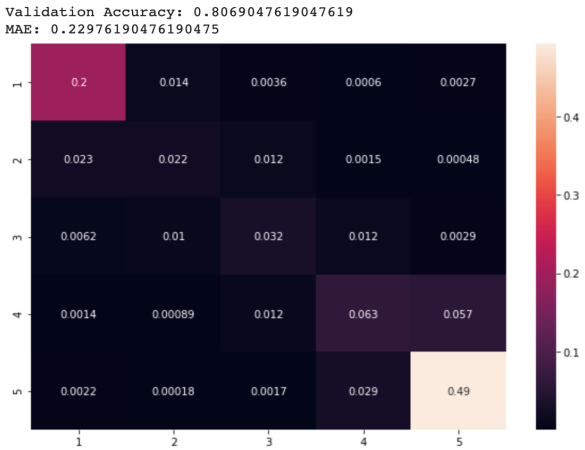


Figure 7: Metrics for Production Model

The top-left and bottom-right diagonals of the confusion matrix show that the model clearly learned to

recognize 5-star and 1-star reviews. Moreover, the MAE of 0.22 stars shows that the model is rarely more than 1 star off when it is wrong, which is a good sign. The model still has room to improve in the 2-4 star range, which not surprising since most of the label noise is likely in this range.

4 Deployment

The majority of my time (60-70%) spent on this project was taken in figuring out how to deploy the model.

TLDR for the gray text below: I spent a long time to figure out that my model was too slow in production on a CPU. I then spent an equally long time to find the fix, which ended up being as simple as:

```
import torch
torch.set_num_threads(1) # single threaded execution
model = ... # Instantiate model
# Quantize model
model = torch.quantization.quantize_dynamic(model, {torch.nn.Linear,
    torch.nn.LayerNorm, torch.nn.Embedding},dtype=torch.qint8)
```

At first, I wanted to challenge myself and deploy the model as an AWS Lambda function since I already have experience with web server deployments. Deploying to AWS Lambda was surprisingly easy, but I quickly ran into issues with the Lambda function timing out. At this point, I did not think the model was a source of latency issues since running one review through the model locally took around 30 ms. Since my model includes a pre-trained component that must be downloaded one time when the Lambda function is first deployed, I thought this was the operations causing the function to timeout. I tried speeding up the download by packaging the pre-trained weights file into the Docker container used to construct the Lambda function environment, but I ran into file permission issues.

I then spent another day or two trying other AWS services (Lightsail and Elastic Beanstalk), but I ran into resource allocation issues with unhelpful error messages and little help from Stack Overflow. I was never able to even deploy a model on these services. Finally, thanks to another student in the class, I finally found a easy-to-use hosting service in Render.

After successfully deploying to Render, I found that sending requests to my prediction endpoint would return a 502 Bad Gateway error. After a few hours (figuring out why my print statements weren't printing), I found that the cause of this error was indeed the forward pass of my model that was timing out. I spent some time researching how others have made inference faster and found that model distillation, quantization, and decreasing input sizes were the three most common approaches. I was already using a distilled model and my input size was already as small as it could get (batch size of 1 with no extra padding), so I tried quantization. For almost a day, I researched why quantization was having no effect on model latency and finally I came across a follow up post deep within a Pytorch issue suggesting restricting Pytorch to single-threaded execution. Implementing this fixed all my deployment issues.

To obtain a prediction from the deployed model, you can send a cURL request as shown below. You should get a response in 100-300 ms, depending on the cache state on the server CPU.

```
curl --location --request POST 'https://yelp-model.onrender.com/predict' \
--header 'Content-Type: application/json' \
--data-raw '{
        "review": "I love this restaurant."
    }'
```

5 Next Steps and Conclusion

The next steps for this project are to return to model training and try out the experiments I was hoping to try:

- **2 Prediction Heads:** Since the classes are ordinal, I think the model could benefit from having both a regression and classification head. I originally included the MAE metric with the intention of trying this model architecture.
- **Different Labels:** To give the model an extra sense of the ordinality of the classes, I want to try a special label encoding. For example, instead of the label for a rating of 2 being $[0, 1, 0, 0, 0]$ (one-hot encoded), it could be $[1, 1, 0, 0, 0]$. With this label encoding, the cross-entropy loss of a prediction with most of its weight on a rating of 3 would be less than a prediction with most of its weight on 5. I think this would give the model enough signal to learn the ordinality of the classes.
- **Hyperparameter optimization:** I want to take advanage of the ease of Weights and Biases sweep feature to experiment more with the learning rate schedule hyperparameters.

In this project, model deployment was the big learning experience for me. With the knowledge I gained from getting my hands dirty and trying to deploy a deep learning model, I now understand why model inference is a large area of research in the lab I am part of; in the absence of careful engineering, efficient model inference at scale would require lots of money. Moreover, going through the motions of deploying a model has made me a lot more aware of the challenges of planning and executing a deep learning project from start to finish (as opposed to stopping after model training as I have done in all of my ML/DL projects until now).