**Week 12 Writeup -**

**Ryan O’Hara, Jack Metzger**

**Pickle the Model**

In Python, *pickling a model* refers to the process of serializing and deserializing a trained machine learning model using the built-in **pickle** module. Serialization transforms the model object—including its weights, parameters, and training state—into a byte stream that can be saved to a file (geeksforgeeks.org). This makes it easy to reload later without retraining, saving both time and computational resources. Pickling our song recommendation model served two main purposes: making the model portable (allowing easy sharing across machines) and reusable (not having to re-train the model for each use). By making the model more portable and reusable, we’ve prepared it for deployment for either industry or public use (through python scripts or a web app).

A white background with text

AI-generated content may be incorrect.

**Figure 1:** Screenshot of the pickling code chunk.

**Environment Dependencies**

To ensure consistent environment setup across machines, we created a “requirements.txt” file listing all the dependencies required to run our song recommendation model. This allows seamless installation of necessary packages for deployment or team collaboration. To create a “requirement.txt” file we activated the environment in the Anaconda Prompt and ran the following line: *pip freeze > requirements.txt*.

To create an environment for this model, create a new environment on a Python command prompt and then run the following command to download all dependencies: *pip install - r requirements.txt.* Our requirements.txt file can be found on our GitHub.

**Deployment**

Our model is an interesting case where it may benefit from different deployment types at different stages in its production lifecycle. First we will describe how we would conduct batch deployment and real time deployment for our specific model, and then we will note how each could potentially work throughout the production process.

First, in the context of batch deployment, we would have to create and train a new model for a certain time period - then given a new influx of data we would retrain a new model and it doesn’t automatically update. In order to do this, we likely would initialize a massive matrix of recommended songs for every song in our dataset, giving us essentially a 3000 by 5 matrix of recommendations after we run the model. We would field input songs from users over a certain timeframe and for every song a user inputs that isn’t currently in our dataset we would collect the metadata and lyrics for the next iteration of the model. The model would not improve over the timeframe, however it would be simple to roll the model out and would see massive gains in improvement from iteration to iteration.

Second, we can utilize a real-time inference deployment, which would be a quick learning and fast updating model that would be ideal when we have a lot of data collected and our model is more mature. Under this deployment type, the model would process and run in real time, making it a slightly longer runtime to garner the recommendations, however this time should be negligible. It would be more difficult to include new songs into the dataset, as it takes a decent amount of time to download and process all of the data, however this would mean the model is learning iteratively and we wouldn’t have to take it down to retrain as often.

Overall, I think initially our group would benefit from a batch deployment, meaning that users could get fast and accurate recommendations and our model could keep getting new song data to improve upon recommendations for each iteration. We could train the model, get all the recommendations and have a recommendation system up and running quickly. Then, after many iterations and our pool of songs grows to a multiple of what it currently is, then there will be less of a need to keep adding songs and retraining the model completely. At that point we would benefit from an automatically updating model in real-time.

**Monitoring Metrics**

Our group would want to monitor multiple metrics both from a model improvement point and a business perspective. We believe that the success of the business model directly ties in to the success of the model itself. As the model gets better, business will naturally grow. This product oriented approach leads us to be focused heavily on improving the model from user feedback, as that will inherently correlate to a business metric.

In improving the model, we will take our biggest performance metric as user feedback. At the end of the day, our model is a tough one to measure accuracy is a quantitative objective manor, as music tastes are independent of each other and entirely subjective. For that, the largest resource we can use to improve our model is the opinions and feedback of others. We plan to leverage this feedback heavily in an attempt to train the model in a somewhat supervised way.

We can conduct this survey simply in two different ways, with one being more complex (but more useful) than the other. First, we can install an easy “good” or “bad” button that easily measures if the song recommendation was quality or not. This is how we have graded our own model in the past, but leaves room for improvement, as some questionable recommendations can go either way and could benefit from a broader scale. For this reason, we think a one through five scale would give a great feedback report so our model gets better feedback. This will give our model a true accuracy score that can be the basis for hyperparameter tuning and overall tuning of the model itself.

From a business standpoint we will attempt to measure retention and time spent with our model for each user. We will be able to tell a lot about how the users are feeling about the product through their rating they give the song recommendations, but there are other ways that we will be able to measure the success of our overall website / application. First, we can measure the amount of time spent asking for recommendations for each user. As the business grows, we want our users to spend an increasing amount of time with the platform.

Another business metric that we can use to see if our users think our service is useful is measure the rate at which they come back to the website after using it once. If we see that a specific user comes to use the model only once and doesn’t come back, it shows they likely didn't enjoy it. If we see improved retention, it shows that people enjoy the recommendations they are getting and it's a quality service.

**Thresholds & Risk Mitigation**

In order to make sure we have a system in place to monitor and maintain a proper working model, we will implement a green, yellow, red system. This system works to establish guidelines that clearly show when our model is working properly, when the model is in danger of being taken down, and when we definitely need to take the model down and assess what is going wrong.

We will have a couple different metrics that will factor into our thresholds to ensure proper risk mitigation. Our model relies heavily on the input of others' opinions, being that it is a subjective accuracy score we are attempting to achieve, so this will be the main thing that goes into our thresholds. It is also important to understand how long the model is taking to run, and while this won’t be an issue when we implement and deploy our model in batches at the beginning, when we pivot to real-time deployment, this will be a key metric to evaluate so our users aren’t waiting extensive amounts of time to get their recommendations.

Our current accuracy scores are hovering around 60-65%, and while we expect that number to grow as we further iterate our model, for now we will use this as a solid accuracy. Also, as mentioned above, recommendation time will not factor in until we use real-time deployment. Based off this standard, the chart below shows how we determine when our model is at green, yellow and red levels:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Green** | **Yellow** | **Red** |
| **User Accuracy** | 55% and up | 40% - 54% | 39% and below |

When the model is green, that means the risk of a poor model has been properly mitigated and there is no concern to be worried about. This threshold of 55% accounts for the variance that we might experience from the 60-65% we would expect to be average. Yellow means we need to start monitoring the model and look for potential warning signs. We would investigate the instances of bad recommendations and try to classify / bucket them to see if bad recs happen more commonly in certain genres or with certain feature characteristics. The model wouldn’t have to be pulled yet, but definitely monitored. Finally, if the model accuracy dips below 40% we would need to pull the model from production. At this point we are giving out more bad recommendations than good ones and a serious analysis of what is going wrong needs to be done to correct our model.

**Model Retraining**

We would plan to initially retrain the model once every two weeks. With our initial user base likely being small, we think if we retrained in shorter intervals we wouldn’t have enough new song data to really alter the model. As the user base grows and more and more songs are being added to our database, we could see ourselves moving this to once every week. Another approach that might work is to retrain the model every X amount of new songs that get suggested. We know that we will add user grading into the retraining, but a big part is seeing what songs users input so we can add to the dataset if they aren’t in there already. If we retrain the model every time 100 new songs are suggested, that might be a better way to approach the timing aspect. We could also do a similar thing for the amount of grades we get on our recommendations.

**Drift**

Data and concept drift are going to be very important to our model, as we plan to not only add data to our model as time goes on, but we don’t know how our users' song preferences may alter throughout time. Data drift will specifically be more important in the short term as we will constantly be adding and changing our model to increase the amount of songs in our database.

Data drift can happen when users input songs that are outside our database and with songs that don't have the proper data needed to be embedded into our matrix. In order to handle this we would have a couple ways to mitigate the risks associated with data drift. First, we would frequently update our song database so if a user requested a song and we didn’t have it, they could come back after a short period of time and that song would be available. Second, we could handle these unknown songs by suggesting something semantically similar to the title / artist of that song or just suggest songs that are the most popular.

We might also see concept drift as the tastes of our users change over the years, where users begin preferring new types of music as genres are ever evolving. As preferences change, it may be hard for our model to recognize over time changes, one way to combat this is to weigh more recent user grading more heavily than old user grading. This would make sure that recent user preferences get factored in to reflect the current state of music taste without completely disregarding historical data we collected.