Byron Washington

CDS 302

May 1st, 2025

Final Project Report

Database Description

This database would serve as a repository for machine learning models, datasets, and algorithms. It would allow users to store, track, and compare different models efficiently. The database would include models, their base model (algorithm), the datasets on which they were trained, performance logs, and the metrics and hyperparameters for those logs. The intended users would be professional or amateur data scientists, machine learning engineers, or analysts.

A function that is expected by users is registering different models. Users must enter the name, version, description, accuracy (in %), and whether it is active when registering a new model. Version is there to allow users to update models, meaning they would have to use and be trained on the same algorithm and dataset, respectively. The description attribute is stored as TEXT in case users want to explain the model in-depth. Unlike other attributes in the table, this is the only attribute allowed to be NULL. Accuracy was chosen as a percentile for the user to easily sort and filter through models and find the best or worst. There is a CHECK constraint on accuracy to ensure it is between 70% and 100%. Any model under 70% accuracy performs poorly, so it shouldn't be entered in the first place. The is_active attribute serves as a flag that determines whether the model can be trusted. It could be inactive because of situations like model decay/drift, old version, or issues with the dataset. In addition to those base attributes, the users would have to enter the model's base algorithm used and the dataset it was trained on. This would mean entering the base_ID and data_ID as foreign keys to those tables. These foreign key constraints hold the relationship between the *algorithm* table, the *dataset* table, and the *model* table. The primary key for *model* is an auto-incrementing integer model_ID, so it does not need to be entered manually unless that's intended. This feature was implemented for ease of use so users wouldn't have to find a free ID when inputting a new model or model version. A scenario where that might occur is if a row gets deleted. The auto-increment function only increases, so there will be a gap, and users might want to fill it in for whatever reason. While having base_ID, data_ID, and name as this table's primary key could work, this would make it harder to insert logs in *performance_log*. For that problem, only model_ID was chosen as the primary key for user ease of use.

The *dataset* table allows users to enter the datasets that models are using. Users must enter the name and a description of the dataset. The name attribute cannot be NULL, but the description is allowed to be NULL if that's what the user intends. The primary key for *dataset* is an auto-incrementing integer data_ID, so it does not need to be entered manually when inserting. This was also implemented for ease of use for the user. An expected function could be if a dataset has an issue, then models trained on it could be set to inactive, letting users know not to use or trust their outputs. Since this database is expected to be publicly hosted, none of the datasets should contain sensitive information, meaning there shouldn't be any privacy concerns.

The *algorithm* table allows users to enter the algorithms being used for each model. Users must enter the name, description, and type of algorithm it is. Description is allowed to be NULL as most algorithm names describe how they work sufficiently. The rest of the attributes are not allowed to be NULL. The type should be 'supervised' or 'unsupervised' to indicate supervised or unsupervised learning. This requirement is enforced with the CHECK constraint. Supervised learning consists of regression and classification because the target value is known and has a label/unit. Unsupervised learning consists of clustering and association rule mining because there is no target value. Typically, it's harder to find the accuracy for unsupervised learning algorithms, especially association rule mining, but this is kept here for the future if there is a metric that is discovered to test the accuracy of those algorithms easily. The primary key for *algorithm* is an auto-incrementing integer base_ID, so it does not need to be entered manually when inserting. This was also implemented for ease of use for the user. An expected function would be a user wanting to see what models are used for supervised learning (classification or regression). Joining *algorithm* and *model* would allow the user to go through those models and decide which one(s) would work best for their specific problem.

In the "base model" relationship in Figure A.1, each algorithm can be a base model for multiple models, but each model only has one algorithm as a base model. For the "trains" relationship in Figure A.1, each dataset can be used to train multiple models, but each model can only be trained on one dataset. These relationships ensure that every model must have exactly one algorithm and dataset attached to it.

For the "based on" relationship in Figure A.1, each model can have many performance logs, but each is based only on one model. Additionally, log_ID is an auto-incrementing integer, so it won't need to be entered manually into *performance_log*. Model_ID cannot be NULL to ensure that logs maintain that "based on" relationship. *Hyperparameter* and *metric* are weak entities to *performance_log*, meaning they need both the log_ID and their name as a primary key. While log_ID doesn't need to be entered into *performance_log,* it is highly recommended to enter it into *hyperparameter* and *metric* to ensure accuracy when logging their names and values, which are not allowed to be NULL. For their relationship, each performance log can hold at least one of each hyperparameter and metric, but each can only be associated with exactly one performance log. *Hyperparameter* and *metrics* are designed as weak entities to allow multiple hyperparameters and metrics for a performance log.

The relational diagram in Figure 1.B was constructed in a way that complies with the BCNF form. BCNF ensures integrity and efficiency within a database when converting from the ER diagram to a relational diagram. Each table within the relational diagram can be proven to be in BCNF form. For *algorithm*, name, description, and type depend entirely on its primary key base_ID. For *model*, name, version, description, accuracy, and is_active are entirely dependent on model_ID. Base_ID and data_ID are only foreign keys for *model*, so none of its attributes are determined by them, complying with the BCNF form. For *dataset*, name and description are entirely dependent on data_ID. For *performance_log,* the only other attribute is its foreign key model_ID. Thus, this table is also in BCNF form. Since *hyperparameter* and *metric* depend on *performance_log*, their attribute “value” depends entirely on their primary keys log_ID and name.

Database Analysis

In the first query, to analyze the database, I looked at all models and sorted them by their accuracy. The highest-performing model in the dataset was “RF-MNIST” version 1.0, a Random Forest model applied to the MNIST dataset, with an accuracy of 96.10%. I also queried the number of performance logs for that model with the highest accuracy. This model also had multiple performance logs, indicating frequent experimentation and optimization. Several other models also surpassed the 90% accuracy threshold, such as “LogReg-Iris” version 2.1 (92.28%) and “NB-Cancer” version 1.2 (91.07%), suggesting that supervised learning algorithms like Logistic Regression and Naive Bayes, when tuned correctly, can offer robust performance on datasets like Iris and Breast Cancer.

In the following query, I filtered the models whose accuracy exceeded the average across the database. These results revealed six models above the average threshold, with the top three being Random Forest, Logistic Regression, and Naive Bayes, like the first query. Further exploration of the fifth-ranked model revealed that it is inactive, meaning we cannot trust its results. However, on average, the fourth and sixth models are only 1% worse than the third-place model, meaning all five models are very effective.

Supervised algorithms dominated model development in exploring algorithm usage, accounting for most models in the database (72%). All algorithms were used at least once, but the supervised algorithm models boast a higher average model accuracy of 88.24%. In contrast, unsupervised algorithm models like K-Means and DBSCAN were less frequent and yielded lower overall accuracies, averaging 81.82%. For example, “KM-Iris” version 1.1 had an accuracy of 88.92%, and “DBSCAN-Iris” version 1.0 scored 76.40%, both respectable but not as high as their supervised counterparts. The caveat is that both those models stated, “KM-Iris” and “DBSCAN-Iris”, are inactive. So, if we wanted only to count models we could trust for unsupervised learning, we can only look at “GMM-Weather”, which has an accuracy of 80.15%.

The next query counted the number of models used for each dataset. The Iris dataset was the most frequently used, and it was associated with three models spanning various algorithms: K-Means, Logistic Regression, and DBSCAN. Interestingly, despite its frequent use, only one of these models exceeded 90% accuracy, and that same model was the only one currently active. This could mean the dataset is flawed and “LogReg-Iris” version 2.1 should’ve been marked inactive, or the more likely reason is that those models haven’t been updated recently and experienced model decay/drift. This term is for when models become less and less reliable over time without tuning because there is often newer data that more accurately reflects the real world. This means the old model won’t be accurate with the newer data because it was trained on too old data and doesn’t reflect reality currently. Within this database, there is no set time when models are ruled inactive after not being updated for a while, but this could be a feature implemented in the future with timestamps for each version. In the next query, I looked at the average accuracy of the models used in each dataset. The MNIST dataset holds the top spot by far, with its model accuracy averaging 96.10%. However, when excluding all inactive models, Iris shot up from 85.87% to 92.28%, leaving it as the 2nd best performing dataset.

I then examined model activity through performance logs. Again, the most logged model was “RF-MNIST”, which had two performance logs, while most other models had one. One model, however, had no associated performance logs, “KNN-Fashion”, possibly indicating either a lack of evaluation tracking or one-time training without follow-up, as shown by its accuracy being below the average for all models. These results show that models with more logs performed better, further strengthening the argument that iterative tuning and evaluation increase the quality of results. I then looked into which types of supervised models performed better. This was done by classifying models that use “f1_score”, “accuracy”, “precision”, “recall”, or “log_loss” in their performance metrics as classification models, while metrics such as “rmse” and “mae”, indicate regression. From these results, I ensured all models were active and took the average for each type of supervised learning. These results showed that, on average, the classification models have a higher accuracy than the regression models (90.34% to 86.88%). However, we must remember that there are four active classification models listed and only two active regression models listed, so if we had more active regression models to look at, they might not be so far apart.

In conclusion, this database analysis shows how effective supervised learning models are, particularly with classification, and emphasizes the importance of regular model evaluation through iterative tuning and reevaluating performance logs. The current schema of the database allows for an easy understanding of how algorithms perform across different datasets. In the future, adding a column that keeps timestamps of each model version could enable users to question whether a model is truly up to date. I don’t know if it’s currently possible to change a feature’s value automatically, but if there is, then that timestamp could serve as a way to verify if that model needs to be retrained/retuned. This database is a valuable tool for managing models, enabling users to make informed decisions based on performance data.

Appendix

Appendix A: Entity-Relationship (ER) Diagram

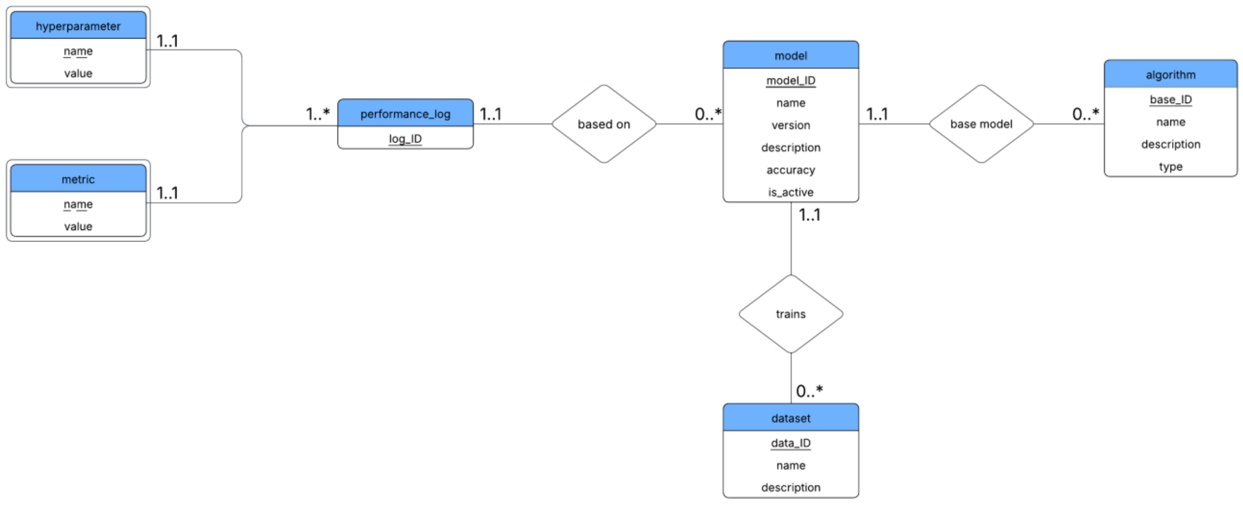

Figure A.1: ER Diagram representing the conceptual model of the database using l..h cardinality format that shows the relationship between entities and their attributes.

Appendix B: Relational Schema Diagram

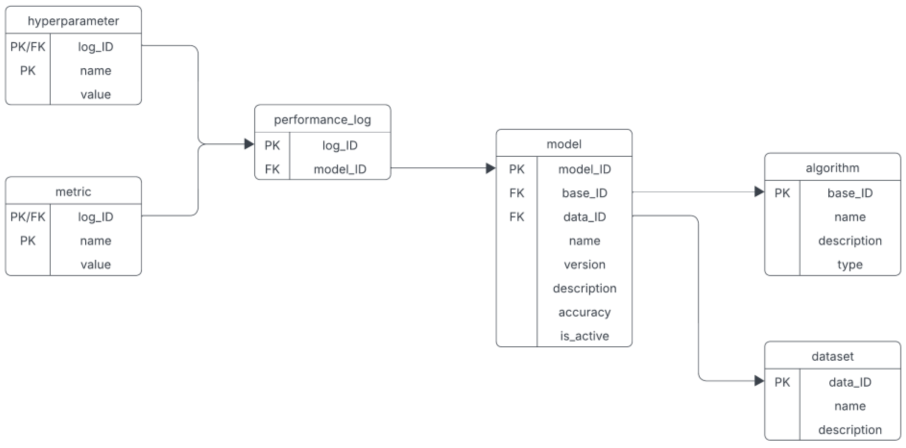

Figure B.1: Relational Schema Diagram derived from the ER Diagram showing tables, labeled primary keys and foreign keys, and foreign keys also with reference lines.