Integrating analytics with relational databases

Raasveldt, M.

CHAPTER 5

In-Database Workflows

1 Introduction

In Chapter 4, we described the inner workings of the MonetDB/Python UDFs. By utilizing these UDFs, existing complex analytical pipelines can be moved inside the database. This allows us to gain all the advantages of storing data inside a relational database, while still having flexible and easy-to-use analytical tools available.

An additional benefit of training and using machine learning models directly in the database is that it is possible to persist both models and metadata (e.g. classification scores on test sets) in the database. Standard relational queries can then be used to apply the trained models to data. This allows for example to compare and combine output from multiple models, each specialized for certain classification tasks. Also, it is possible to classify the same data using multiple models and use the result of the model that reports the highest confidence.

In this chapter, we showcase how we can use MonetDB/Python UDFs to efficiently integrate a complex analysis pipeline inside MonetDB. We show how we can train models directly inside the database, and how to store the models and subsequently use

them to classify data without having to export the data from the database system.

1.1 Contributions

In this chapter, we show how traditional classification models can be integrated into a column-store relational database management system. We describe how models can be stored inside the database system and how these models can then be used to efficiently and flexibly classify data. We experimentally show the performance benefit of directly running the models inside the database system versus loading the data from structured text, binary files or using database client protocols.

1.2 Outline

This chapter is organized as follows: Section 2 discusses related work. Section 3 presents our integration approach, followed by a concrete use-case and performance results in Section 4. Finally, we draw our conclusions in Section 5.

2 Related Work

There is a variety of related work on combining relational database systems with machine learning pipelines. In this section we will present the most recent related work regarding the integration of machine learning through UDFs and model management systems and compare them with our solution.

2.1 Machine Learning Integration

Integrating existing Database Management Systems and machine learning algorithms has been a long standing problem due to the complexity of implementing the machine learning code inside a DBMS.

Early work [73, 3] on this focuses on rewriting analytical algorithms into portable SQL code. This allows the pipelines to be executed within any database system without requiring database-specific modifications. However, rewriting complex analytical

pipelines in SQL requires a lot of manual effort and might not be possible for certain algorithms because SQL is not a Turing complete language.

In Ordonez et al. [63], machine learning algorithms are translated to either C, C++ or C# code (depending on the DBMS language support) and inserted into UDFs. As a consequence they achieve high performance when analyzing large data sets compared to external data analysis tools, as data movement is mitigated. However, these algorithm must be coded in one of the previously listed languages. This often results in the need for rewriting code, because most prominent machine learning libraries are usually available in scripting languages (e.g., Python and R). In our solution we allow the developer to use popular scripting languages *together with their entire ecosystem of data analytics packages* as UDFs in MonetDB.

Other work [26, 17, 37] focuses on more templated approaches for machine learning integration to reduce the necessity of code rewriting. However, the main disadvantage of these methods is that they only work for a limited subset of algorithms, which limits their applicability to general machine learning tasks.

2.2 Machine Learning Model Management

When training and using a variety of models the problem of managing these models arises. This problem is exasperated because most Machine Learning Systems do not provide support for storing and querying their models. Due to these issues, data scientists quickly lose track of their models.

In Vartak et al. [85], a system called ModelDB is introduced that can be used for storing, tracking and managing machine learning models in their native environment. This allows data scientists to use SQL to query their models based on their metadata (e.g., hyperparameters, parameters) and quality metrics (e.g., accuracy). It also has the option to store the used train/test data sets for each model. However, since ModelDB only stores the models in their native environment, it does not provide a solution for coupling machine learning applications with traditional relational databases.

3 Machine Learning integration

Machine learning pipelines consist of three stages [21].

1. **Preprocessing.** In this stage, the raw data is loaded and cleaned. The data is normalized, and any inconsistencies from incorrect or missing measurements are corrected for or removed.

2. **Training and Verification.** In this stage, the cleaned data is used to train the model. Typically the training set is divided into parts, and techniques like cross validation are used to prevent overfitting the model.

3. **Classification.** In the final stage, the trained model is used to classify new data. In this stage, the model can still be refined further based on new data or new properties of the data.

The preprocessing stage can often be performed entirely within traditional database management systems. Loading data and simple cleaning operations such as missing value removal can be done using standard SQL queries. However, when more advanced preprocessing such as interpolation is required, user-defined functions can be used to simplify this step.

The real challenge of integrating these pipelines into databases, however, is implementing the machine-learning models. The models rely on complex math operations and iterative refinement, which are not supported by standards-complaint SQL.

There are many libraries and packages in vectorized scripting languages that implement common machine learning and classification models, such as TensorFlow [2] and Sci-Kit Learn [65]. Using vectorized user-defined functions, we can plug these libraries into the database. However, the typical processing pipelines must be adjusted so they can fit into a SQL workflow. In this section, we will describe how these analytical pipelines can be integrated into traditional database management systems through the use of user-defined functions.

3.1 Training

To train a classification model, we take a set of annotated data as input and use the annotations to find patterns in the data. After learning these patterns, the trained model can accurately classify un-annotated data.

The training pipeline therefore takes as input a set of columns representing the data, and a single column representing the classes of the data. This will be the input to our user-defined function. The output of this stage of the pipeline is the trained model, which will be the output of our UDF. The actual creation and training of the model will happen inside the function.

Model Storage. Models exist as in-memory objects within the scripting language. However, they can be serialized to a binary format for persistent storage on disk. In Python, this is done using the `pickle` library. In order to store the objects in the database we need to serialize the objects to this binary format, after which we can place them in a `BLOB` field.

```
CREATE FUNCTION train(data INTEGER, classes INTEGER,
    n_estimators INTEGER)
RETURNS TABLE(classifier BLOB, estimators INTEGER)
LANGUAGE PYTHON
{
    import pickle
    from sklearn.ensemble
        import RandomForestClassifier

    clf = RandomForestClassifier(n_estimators)

    clf.fit(data, classes)

    return {'classifier': pickle.dumps(clf),
        'estimators':n_estimators }
};
```

Listing 5.1: Training The Model

An example of a user-defined function that trains a Random Forest Classifier using Sci-Kit Learn is given in Listing 5.1. This is a vectorized user-defined function, and as such both `data` and `classes` are vectors of integers within the function instead of individual elements. This function can be called from within SQL with the model data, classes and the amount of estimators (i.e., model parameters) as input, and will produce a table containing the trained classifier and its meta-data as output. This table can either be stored in the database, or used directly as input to another function that uses the trained classifier (if no persistent storage is necessary). Note that it is trivial to alter this UDF to train a different model from the Sci-Kit Learn library, as all that is required is importing a different model and using that.

3.2 Classification

After the model has been trained, it is ready to accept unlabeled data and can be used to classify that data. The classification stage therefore takes as input a set of columns representing the unannotated data, and the trained classifier that will be used to classify the data. The output is the set of predicted labels produced by the classifier. Inside the user-defined function, the classifier will again have to be deserialized into an in-memory object, after which it can be used to classify the input data and produce a set of labels.

```
CREATE FUNCTION predict(data INTEGER, classifier BLOB)
RETURNS INTEGER
LANGUAGE PYTHON
{
    import pickle
    classifier = pickle.loads(classifier)
    return classifier.predict(data)
};
```

Listing 5.2: Classification

An example of a user-defined function that classifies a set of data is given in Listing 5.2. This function can be called from within SQL with the unlabeled data and

the classifier as input, and will produce a list of predicted classes.

The predict function can be used both to test a trained model and to classify a set of new data using such a model. The model can be tested by predicting a set of data for which the labels are known, and comparing the predicted labels against the new labels. The model can be used to

3.3 Ensemble Learning

In addition to only storing the trained models, we can store additional metadata about the models in the database. This metadata can include information such as parameters used to instantiate the model, or information about the effectiveness of the model obtained through testing it against certain datasets. We can then choose a model to classify new data based on this metadata, or we could classify the data using multiple models that are stored and use the results from the classifier with the highest confidence.

4 Experimental Analysis

In this section, we demonstrate how a real classification pipeline can be integrated into a column-store database, and show how the in-database processing pipeline performs when compared against the same pipeline implemented in a standard scripting language where the input data is loaded from a file or transferred over a database socket connection.

The pipeline we use in our experiments is used to attempt to classify who people from North Carolina will vote for in the Presidential Elections based on data from the 2012 Presidential Election. For this purpose, we use two separate datasets:

- **The North Carolina Voters Dataset** contains the information about the individual voters. This is a dataset of 7.5M rows, where each row contains information about the voter. There are 96 columns in total, describing characteristics such as place of residence, gender, age and ethnicity. Note that we do not know

who each person actually voted for, as this information is not publicly available.

- **The Precint Votes Dataset** contains the aggregated voting statistics for each precinct, (i.e., how many people in each precinct voted Democrat, and how many voted Republican). This dataset has 2751 rows, one for each precinct in North Carolina.

By combining these two datasets we can attempt to classify individual voters. We know the voting records of a specific precinct, and we know in which precinct each person voted, so we can make an educated guess who each person voted for based on this information.

Preprocessing. As we do not have the true class labels for each voter, we have to generate them from the information we have about the precincts. This requires us to join the voter data with the precinct data, giving us the voting records of the precinct that each voter voted in. We then generate a "true" class label for each voter using a weighted random function based on the precinct voting records. For example, if voters in a specific precinct voted for Democrats 60% of the time, each voter in that precinct has a 60% chance of being classified as Democrat and 40% chance of being classified as Republican.

Training. After we have generated the true class labels, we have to train the model using the data and the labels. However, we don't simply want to use all the data for training. Instead, we want to divide the data into a training set and a test set to prevent overfitting. We then feed the data in the training set to the model using the function shown in Listing 5.1 and store the resulting model in the database.

Testing. After the model is trained, we want to test how it performs by classifying the data in the test set and looking at the results. We can classify the voters in the test set by running the function shown in Listing 5.2. After having obtained the predicted class labels, we can test the accuracy of our model by comparing against the known true class labels of the data. However, since we only have the generated class labels of the individual voters, comparing the predicted labels against those would not give us a lot of information about our classification accuracy. Instead, we aggregate

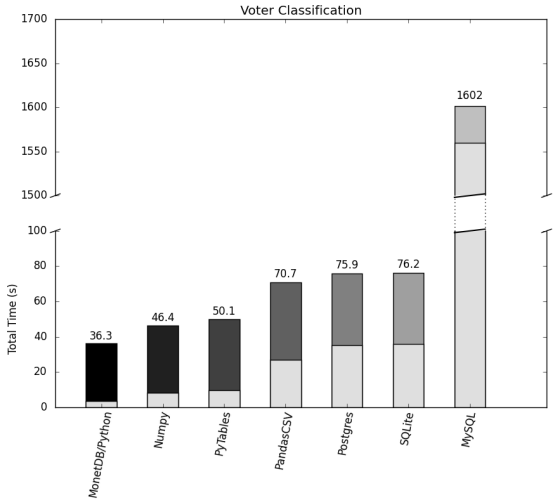

Figure 5-1: Voter Classification Benchmark

the total amount of predicted votes for each party by precinct. Then we compare the aggregated predictions against the known amount of votes in each precinct.

Performance Analysis. To determine how well our in-database processing solution performs compared to ad-hoc analysis pipelines we have implemented the pipeline described above both (1) using MonetDB/Python UDFs and (2) inside Python, using various different methods of initially loading the data. For loading the data in Python, we have experimented with loading from binary files (NumPy [84] files and HDF5 [80] using PyTables), CSV files using an optimized parser, transferring the data to Python through a database socket connection (with PostgreSQL [77], MySQL [89] and SQLite [4] as database servers). For the scenarios where the data is stored inside a relational database, we use SQL to perform the preprocessing steps involving joins and aggregations. Whereas for the pure Python solutions, we use the Pandas library [56] to perform these steps.

The experiments were run on a Fedora (Release 26) machine with 2.6GHz 8-core Intel Xeon processor (Turbo Boost up to 3.2GHz), 20MB shared L3 cache and 256 GB of RAM. All the tests are hot runs. The datasets and source code used for the experiments are publically available[1].

[1] `https://github.com/pholanda/VoterClassification`

Results. The results of the benchmark are displayed in Figure 5-1. The numbers display the total time required to run the entire classification pipeline, whereas the bottom gray bars indicate the time spent loading the initial data into Python and performing the initial preprocessing steps and aggregations.

We can see that the in-database processing solution using MonetDB/Python is significantly faster than the alternative database solutions. The time spent on initial wrangling of the data is an order of magnitude lower than transferring it over a socket connection using the other database solutions. We also note that loading the data from CSV files is comparable in speed to transferring the data over a socket connection.

Loading the data from binary files is much faster than loading from structured text or transferring the data over a socket connection. However, this introduces additional challenges in managing the data. Especially in the case of NumPy binary files, where each of the 96 columns is stored as a separate file on disk. We do still see that the in-database processing solution spends less time on initial wrangling of the data and runs the entire pipeline significantly faster.

5 Summary

In this work, we have shown how complex analysis pipelines can be efficiently integrated into column-store databases. Using these pipelines, it is possible to perform preprocessing, training, testing and prediction using complex machine learning models directly on data stored within a relational database. We have demonstrated the efficiency gained from using these in-database processing methods, and shown the additional benefits that come with storing data in a relational database system.