Engine of Automatic Machine Learning

CPSC 490 Project Report

Felipe Pires

Advised by Kyle Jensen

December 12, 2018
Abstract

As of 2018, many open-source libraries exist out there that solve specific problems in machine learning very well, at least when operated by programmers with a good understanding of the mathematics that underlie them. Despite all the resources being poured into artificial intelligence through these libraries, few tools exist that solve the simple data science needs of users lacking a strong technical background. In developing this project, we set out to capitalize on these high-quality open-source libraries to solve simple machine learning tasks with very little configuration.

In this project, we set out to build a program that could, with a high degree of autonomy, execute tasks that data scientists are commonly in charge of, such as training and deploying machine learning models to predict labels in datasets, finding anomalies in spreadsheets etc. Equipped with such a tool, programmers would be able to create applications that automate part of the typical data science workflow and help teams with little technical background profit from recent advances in artificial intelligence. With this use case in mind, we set out to build our solution as a service: a program that runs on a remote server and executes jobs it receives from a messaging queue. In order to solve machine learning problems autonomously, our goal was to capitalize on existing high-quality data science libraries using whichever library was most suited for the task at hand.

In the end, we succeeded in developing a service that is capable of executing classification and regression tasks given any tabular data, and used it to solve competitions on the website Kaggle.
Database Design

A few choices had to be made before we could start programming, such as which languages to write our code in and where to store our datasets. For the language, we decided to use Python, which is a very popular language for data science in 2018. Python’s popularity in the data science community meant that the most important machine learning libraries are either ported to or directly written in Python, which was an attractive feature for the purposes of this project. For storing datasets, we decided to use Amazon’s AWS S3. We realized early on that, in order to build a reliable service that interacts with the world through a messaging queue, our datasets would have to be stored in a remote storage system that both our clients and our service have access to. The alternative, ie. having clients send datasets to our service as part of the messages, would severely limit the size of datasets that we could work with.¹ So we decided to store our datasets in an S3 bucket, which we gave both our clients and our program access to.

Data versioning with snapshots

One key requirement that helped guide our database design was having datasets be immutable, so that old datasets could always be recovered even if our program misbehaved. Data science work tends to be full of trial and error, and immutable datasets would allow us to move fast on developing features, knowing that backups would always be at hand in case we corrupted data because of untested code.

¹ For instance, the messaging queue we used for this project, RabbitMQ, limits the total size of its queue to 1MB. [https://www.rabbitmq.com/maxlength.html](https://www.rabbitmq.com/maxlength.html)
This requirement of data immutability inspired the creation of two separate components to describe a dataset: bases and snapshots (see Figure 1). **Bases** encapsulate the mutable aspects of a dataset. Each contains a name (eg. “customers”), a description (eg. “List of all company customers.”), an owner and a number of other attributes. **Snapshot** objects, however, are immutable; they contain a snapshot of a dataset at a certain time. Each has an s3path attribute, pointing to a file in S3 containing the data stored in JSON format. They also have a features attribute, describing the different columns of the dataset stored in S3, and base and version attributes, which together must make a unique pair.

Figure 1. Database schema

This design helps us deliver immutability through data versioning. Snapshots are read-only objects. In order to change the data that a base points to, our program...
must create a new snapshot object, with an updated \texttt{version} number, and linked to a new file on S3. It must also increment the base’s \texttt{snapshot_version} accordingly. As long as these conditions were followed, our service would never overwrite important information.

\textbf{Insight and model objects}

An \texttt{insight} represents a machine learning task, which a client wants executed on a certain snapshot of data. For example, given a snapshot of insurance customers, a client might want an insight that predicts which customers are more likely to file an insurance claim in the future. Once an insight is created, our program needs to train the necessary statistical models using Python and store these models in S3 so they can be used to make predictions at some later time. For instance, to create a classification insight, our program train XGBoost tree booster, then serializes them as \texttt{pickle} files and stores them in an S3 bucket.\textsuperscript{2} Afterwards, in order to run these boosters, our script fetches the pickle files from S3 and deserializes them back into tree booster objects in Python. All the information required to fetch and run these boosters is encapsulated in a \texttt{model} object and saved to the database.

Like bases and snapshots, insights and models also support versioning through the \texttt{insight.version} and \texttt{model.insight_version} attributes. Separating mutable from immutable information as insight and model objects helps us ensure that previous machine learning assets are never lost, even if our service misbehaves.

\textsuperscript{2} \url{https://docs.python.org/3/library/pickle.html}
Autonomous Layer

As of 2018, many open-source libraries exist out there that solve specific problems in machine learning very well, at least when operated by programmers with a good understanding of the mathematics that underlie them. The idea which prompted the development of this project was the observation that, despite all the resources being poured into artificial intelligence through these libraries, few tools exist that solve the simple data science needs of users lacking a strong technical background. In developing this project, we set out to capitalize on these high-quality open-source libraries to solve simple machine learning tasks with very little configuration.

Figure 2. A log showing example message received from the queue server

We built our program to carry out two types of data science tasks: profile datasets and train/run predictive models. Clients send these tasks to our service using a
remote messaging queue service, running RabbitMQ (see Figure 2). For each task, our program creates a corresponding job object, which it uses to communicate progress with the client. Insights are then sorted out inside src/demux.py and the appropriate handlers are called in src/jobs/. Once the handler finishes executing or throws an exception, the job status is changed to “FINISHED” or “FAILED” and a JSON response is sent back to the client through the queue.

Solving classification and regression

To carry out classification and regression insights, we elected to use XGBoost, a powerful library for creating gradient boosted tree models. We chose XGBoost in part due to our previous familiarity with random tree models, which facilitated initial hyperparameter tuning, but also due to the library’s current popularity within the data science community, which made it easy to find tutorials explaining how to use it.

```python
xgbParams = {
    'max_depth': 10,
    'learning_rate': 0.01,
    'silent': 1,
    'objective': 'multi:softmax',
    'num_class': 2,
}
numRounds = 20
cv = xgb.cv(xgbParams, dtrain, numRounds)
model = xgb.XGBClassifier(**xgbParams)
LOGGER.info('Starting training')
model = xgb.train(xgbParams, dtrain, numRounds, verbose_eval=True)
LOGGER.info('Finished training')
```

Figure 3. Code excerpt from src/ml/XGBoostClassifier.py

---

3 https://www.rabbitmq.com/documentation.html
Configuring XGBoost was straightforward. For classification, we only had to tweak two parameters: the step size for learning and the max depth of the boosted trees model (see Figure 3). Part of a data scientist’s job, in this case, would be to tune these two parameters and achieve the desired balance of bias and variance for the model. Too little depth and the model will “underfit”, meaning the variance is needlessly high. Too much depth, on the other hand, causes the model’s bias to be needlessly high, which is called “overfitting”. Through experimentation, we settled on a tree depth of ten and a learning rate of 0.01. Moving forward, it would be a good idea to change these hyperparameters dynamically using a search library like hyperopt.⁵

Data treatment

Training and running models involves a good amount of data processing, which we accomplished using the pandas and numpy Python libraries.⁶ Our handlers are programmed to fetch the dataset for each task from their appropriate location in S3. The datasets in S3 are stored in JSON format, and deserialized by the handler into a Pandas “dataframe” object. The most important processing we do is one-hot encode the categorical variables in a dataset before we feed it to tree boosters.⁷ This and other data treatments are programmed in src/processors.

---

⁵ [https://github.com/hyperopt/hyperopt](https://github.com/hyperopt/hyperopt)
⁶ [https://github.com/pandas-dev/pandas](https://github.com/pandas-dev/pandas), [https://github.com/numpy/numpy](https://github.com/numpy/numpy)
⁷ We found no consensus on the practice of one-hot encoding variables. See [https://datascience.stackexchange.com/questions/9443](https://datascience.stackexchange.com/questions/9443), [https://medium.com/data-design/53400fa65931](https://medium.com/data-design/53400fa65931).
Profiling models

Every time it trains a model, our program also uses XGBoost’s cross-validation routines to provide indicators of model performance to the client. These indicators are stored in a separate model_profile object. Examples of statistics we keep track of are the mean and standard errors for the multiclass boosters we create. XGBoost also offers us the “gain” and “weight” measures of feature relevance for boosted trees models, which we also store inside the model_profile object.

Deployment

Our program was set up to work as a service, listening for requests through a remote server running a RabbitMQ. We used AWS S3 to share large files between the client and our service, such as datasets and serialized machine learning models. To host our SQL data, we chose Postgres, one of the most widely used relational database management systems out there.

To facilitate the operation of these services, we chose to use Docker. Docker’s docker-compose toolkit allowed us to easily configure RabbitMQ and Postgres instances for development and deployment (see Figure 4). We tested the reliability of our service by setting it up on a remote machine on AWS EC2. Inside an EC2 instance running Ubuntu 16.04, we used Docker to spin up RabbitMQ and Postgres servers, exposing their ports to the external connection so that clients could access

---

9 Community-curated information about big companies using Postgres is available at https://stackshare.io/postgresql.
10 https://docs.docker.com/compose
them. We then successfully used Docker to package and run our program as a container.

```yaml
version: '3'
services:
  postgres:
    image: postgres
    restart: always
    ports:
      - 5432:5432
    environment:
      POSTGRES_PASSWORD: example
      POSTGRES_DB: project490

  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - 15672:15672
      - 15671:15671
      - 5672:5672
    environment:
      RABBITMQ_DEFAULT_PASS: guest
      RABBITMQ_DEFAULT_USER: guest
      RABBITMQ_DEFAULT_VHOST: project490
    healthcheck:
      interval: 10s
```

Figure 4. The configuration file for Docker Compose

**Web app for test driving**

Creating our program as a service meant that, in order to properly test it, we had to create a client. Concurrent with this project, we also built an application that interacts with our service (see Figure 5).
In order to test drive our system, we decided to apply our system against real-world machine learning problems, which we took from the website Kaggle.\textsuperscript{11} The first problem we solved was the Safe Driver Prediction challenge created by Brazilian car insurance company Porto Seguro (see Figure 6). Using anonymized datasets of hundreds of thousands of Porto Seguro customers, our system automatically created two tree boosters to predict which customers were most likely to file an insurance claim in the future. The resulting models had an average cross-validation mean error rate of 3.70%, which we were very pleased about.

\textsuperscript{11} https://www.kaggle.com/c/porto-seguro-safe-driver-prediction
Conclusion

With our final product, we accomplished most of the deliverables we set out to work on at the beginning of the semester. We started off by using XGBoost to implement classification tasks with fixed hyperparameters. As the weeks progressed, our focus shifted from increasing the variety of machine learning tasks our program could solve, as previously intended, to making our program into a reliable stand-alone service that could be integrated into other applications with ease. Throughout, a lot of attention was paid database and codebase design. Most of the code went through multiple rewritings, which improved clarity and separation of concerns between the different components. Our experiments using datasets from Kaggle had a high degree of success and showed that, to a large degree, relevant machine learning
tasks can already be executed automatically by a system like ours, without the need for data scientists.